目录

开篇、算法秘籍阅读指南

无剑篇、刷题心法

- 学习算法和刷题的框架思维
- 计算机算法的本质

学剑篇、基础数据结构

- 1.1 数组/链表
    - 小而美的算法技巧：前缀和数组
    - 小而美的算法技巧：差分数组
    - 我写了首诗，把滑动窗口算法算法变成了默写题
    - 我写了首诗，把二分搜索变成了默写题
    - 二分搜索题型套路分析
    - 田忌赛马背后的算法决策
    - 一文秒杀四道原地修改数组的算法题
    - 一文搞懂单链表的六大解题套路
    - 链表操作的递归思维一览
- 1.3 队列/栈
    - 队列实现栈以及栈实现队列
    - 一文秒杀三道括号题目
    - 单调栈结构解决三道算法题
    - 单调队列结构解决滑动窗口问题
    - 一道数组去重的算法题把我整不会了
- 1.4 数据结构设计
    - 算法就像搭乐高：带你手撸 LRU 算法
    - 算法就像搭乐高：带你手撸 LFU 算法
    - 给我常数时间，我可以删除/查找数组中的任意元素
    - 一道求中位数的算法题把我整不会了

仗剑篇、进阶数据结构

- 2.1 二叉树
    - 东哥带你刷二叉树（第一期）
    - 东哥带你刷二叉树（第二期）
    - 东哥带你刷二叉树（第三期）
    - 二叉树的序列化，就那几个框架，枯燥至极
    - 美团面试官：你对后序遍历一无所知
    - 东哥带你刷二叉树（总结篇）

- 2.2 二叉搜索树
    - 东哥带你刷二叉搜索树（第一期）
    - 东哥带你刷二叉搜索树（第二期）
    - 东哥带你刷二叉搜索树（第三期）
- 2.3 图论
    - 图论基础
    - 拓扑排序
    - 二分图判定
    - 并查集算法详解
    - Kruskal 最小生成树算法
    - 我写了个模板，把 Dijkstra 算法变成了默写题
    - 众里寻他千百度：名流问题

霸剑篇、暴力搜索算法

- 3.1 DFS 算法/回溯算法
    - 回溯算法解题套路框架
    - 回溯算法牛逼：集合划分问题
    - 一文秒杀所有排列/组合/子集问题
    - DFS 算法秒杀所有岛屿题目
- 3.2 BFS 算法
    - BFS 算法解题套路框架
    - 如何用 BFS 算法秒杀各种智力题

悟剑篇、动态规划

- 4.1 动态规划核心原理
    - 动态规划解题核心框架
    - base case 和备忘录的初始值怎么定?
    - 最优子结构和 dp 数组的遍历方向怎么定?
    - 提高刷题幸福感的小技巧
- 4.2 经典动态规划
    - 最长递增子序列问题
    - 最大子数组和问题
    - 最长公共子序列问题
    - 编辑距离问题
    - 正则表达式问题
- 4.3 背包问题
    - 0-1 背包问题
    - 完全背包问题

    - 子集背包问题

- 4.4 用动态规划玩游戏
    - 团灭 LeetCode 股票买卖问题
    - 团灭 LeetCode 打家劫舍问题
    - 动态规划之博弈问题
    - 动态规划之最小路径和
    - 经典动态规划：高楼扔鸡蛋
    - 动态规划帮我通关了《魔塔》
    - 动态规划帮我通关了《辐射4》
    - 旅游省钱大法：加权最短路径

朴剑篇、其他经典算法

- 5.2 数学算法
    - 如何高效寻找素数
    - 两道常考的阶乘算法题
    - 如何在无限序列中随机抽取元素
    - 东哥吃葡萄时竟吃出一道算法题
    - 如何同时寻找缺失和重复的元素
- 5.3 面试必知必会
    - 一个方法团灭 nSum 问题
    - 一个方法解决三道区间问题
    - 快速排序亲兄弟：快速选择算法
    - 分治算法详解：运算优先级
    - 扫描线技巧：安排会议室
    - 当老司机学会了贪心算法
    - 剪视频剪出一个贪心算法
    - 如何实现一个计算器
    - 谁能想到，斗地主也能玩出算法
    - 如何判定完美矩形

开篇、算法秘籍阅读指南

这本 PDF 是 labuladong 的刷题三件套 中的第一件：《labuladong 的算法秘籍》。

我的 GitHub 算法仓库 目前已经快 100k star 了（疯狂暗示点 star），为了感谢大家一直以来的支持，我制作了刷题三件套。

刷题三件套共包含《labuladong 的算法秘籍》和《labuladong 的刷题笔记》这两本 PDF 以及 labuladong 的辅助刷题插件。

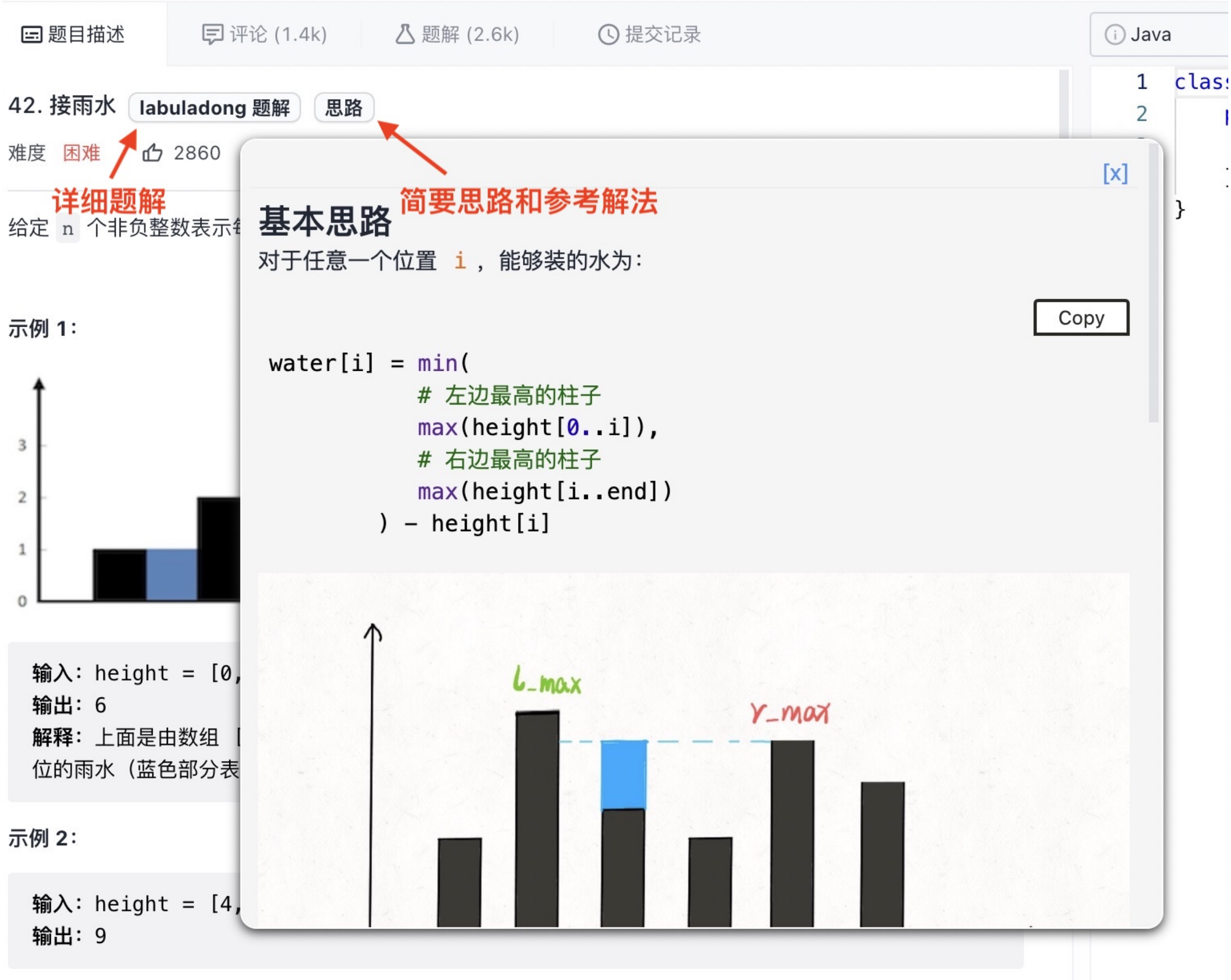
如果你是从我的公众号「labuladong」后台下载的三件套，会发现 **PDF 和插件都有版本号，也就是说我一直在更新三件套的内容**，修正错误或更新内容，所以你应该选择最新版本学习。

这本《算法秘籍》是首先建议阅读的，读完之后可以阅读《刷题笔记》。你可以把《算法秘籍》理解成教材，《刷题笔记》理解成一本练习册，当然应该先看教材，再通过练习册巩固复习。

这本《算法秘籍》会详尽解析各种算法的原理和应用，而《刷题笔记》的灵感来源于「单词速记卡」的形式，其中只列出每到题目简明扼要的思路和参考解法。

你可以在碎片化的时间翻看《刷题笔记》，像背单词一样背算法。至于这本《算法秘籍》，由于内容比较硬核，所以建议拿出整块的时间仔细阅读和思考，并亲手做题实践。

而 labuladong 的刷题辅助插件，完美融合了上述两本 PDF 的内容，能够在力扣题目页面显示《算法秘籍》中对应的详细题解和《刷题笔记》中的简明思路（也支持英文版 LeetCode），是建议每个读者都安装的：



而且插件目前提供了**手把手**带你刷通所有二叉树题目的功能，未来还会添加手把手刷动态规划等功能，详情见这里。

labuladong 的刷题三件套 中提供了 PDF 的下载和插件的使用教程，这里就不多说了。

算法秘籍目录结构

如果是金庸武侠小说的读者，应该熟悉《神雕侠侣》中杨过发现剑冢的情节，孤独剑魔一生练剑分为几个阶段：

第一阶段：青光利剑，凌厉刚猛，无坚不摧，弱冠前以之与河朔群雄争锋。

第二阶段：紫薇软剑，三十岁前所用，误伤义士不祥，悔恨无已，乃弃之深谷。

第三阶段：玄铁剑，重剑无锋，大巧不工。四十岁前恃之横行天下。

第四阶段：四十岁后，不滞于物，草木竹石均可为剑。自此精修，渐进于无剑胜有剑之境。

我给这本 PDF 起名「算法秘籍」是有理由的，我觉得学习算法的过程就好似孤独剑魔练剑：

第一阶段：虽然算法技巧的储备比较匮乏，刷题比较吃力，但每每遇到新的算法技巧，都会大呼精妙，学习的乐趣会抵消挫败感。

第二阶段：对常见的算法技巧都已有了一定的知识储备，却苦于无法自如运用这些技巧，看到一道算法题很难洞悉其本质，无法转化成自己熟悉的题型来解决。

第三阶段：各种算法技巧已比较纯熟，理解到计算机算法的本质即为穷举，看到一道题目，大致就知道要用什么技巧来解决。

第四阶段：随着持续刷题精进，通汇贯通，不只把算法当做面试的工具，进而将算法融入工作和生活，解决实际问题。

这本算法秘籍从这种算法的原理入手，可以帮你走到第三阶段，同时希望你能够爱上算法，持续修炼，达到第四层境界。

《labuladong 的算法秘籍》主要分为基础数据结构、进阶数据结构、暴力穷举算法、动态规划、其他经典算法几部分，

章节编号借鉴了《说剑》这款游戏中的几个关卡：学剑、仗剑、霸剑、朴剑、无剑，每个章节页的图片均来自《说剑》这款游戏的关卡截图，这样似乎更有点「秘籍」的味道了。

最后，我的公众号 labuladong 积累了很多高质量且通俗易懂的算法文章，这本 PDF 中只选择性地收录了一部分。

一方面因为 PDF 页数不宜过多，否则容易把人吓退，另一方面因为不能影响到纸质书《labuladong 的算法小抄》的销售：



算法小抄于 2020 年年底出版，本 PDF 主要收录的是一些经典算法技巧和纸质书出版之后的公众号文章，如果想学习更多算法文章或者购买纸质书，可以关注我的公众号查看。

另外，这本 PDF 的内容也可以在我的公众号查看，目录入口如下：


公众号
labuladong
发消息
致力于把算法讲清楚，刷题也可以很简单。
194 篇原创内容 4269 位朋友关注
视频号：labuladong
#手把手刷动态规划
#手把手刷数据结构
展开
消息
视频号
服务
labuladong 提供的服务
目录
公众号目录
算法秘籍目录
刷题笔记目录
刷题套装
算法秘籍 PDF
刷题笔记 PDF
刷题插件
算法小抄纸质书
组队打卡
取消

后续我会持续输出高质量算法文章，**公众号菜单有我亲自制作训练营和课程以及刷题打卡活动**，大家持续关注公众号的通知即可



另外，也建议关注我的微信视频号，每周我都会抽空直播，而且会在视频号积累学习算法的短视频，分享自己的学习经验：



无剑篇、刷题心法



我每周在视频号直播的时候都有读者问我，如何高效刷题，刷完题目过几天又忘了怎么办之类的问题。

这些问题对于初学者来说肯定是难免的，但如果学了挺久算法还有这种问题，那大概率是方法有问题了。

什么是好的学习方式？能够做到刷一道题懂十道题，举一反十，这就是好的学习方式，不然的话现在力扣两千多道题，全给他刷完的话还干不干别的事情了？

我在公众号经常强调的**框架思维**，就是帮助大家培养举一反三的能力。

做题做错没关系，只要你能够跳出细节，从整体上理解各种技巧的底层逻辑，那不仅下一次能作对，而且再把题目给你变十个花样，你还是能做对，爽不爽？

第一章不会列举什么具体的算法技巧，就是单纯的思维模式，也许对于初学者来说不太能理解其中的奥妙，相信在之后的题目实践中会逐渐体会出一点意思的。

学习算法和刷题的框架思维

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong



学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

这是好久之前的一篇文章 学习数据结构和算法的框架思维 的修订版。之前那篇文章收到广泛好评，没看过也没关系，这篇文章会涵盖之前的所有内容，并且会举很多代码的实例，教你如何使用框架思维。

首先，这里讲的都是普通的数据结构，咱不是搞算法竞赛的，野路子出生，我只会解决常规的问题。另外，以下是我个人的经验的总结，没有哪本算法书会写这些东西，所以请读者试着理解我的角度，别纠结于细节问题，因为这篇文章就是希望对数据结构和算法建立一个框架性的认识。

从整体到细节，自顶向下，从抽象到具体的框架思维是通用的，不只是学习数据结构和算法，学习其他任何知识都是高效的。

一、数据结构的存储方式

数据结构的存储方式只有两种：数组（顺序存储）和链表（链式存储）。

这句话怎么理解，不是还有散列表、栈、队列、堆、树、图等等各种数据结构吗?

我们分析问题，一定要有递归的思想，自顶向下，从抽象到具体。你上来就列出这么多，那些都属于「上层建筑」，而数组和链表才是「结构基础」。因为那些多样化的数据结构，究其源头，都是在链表或者数组上的特殊操作，API 不同而已。

比如说「队列」、「栈」这两种数据结构既可以使用链表也可以使用数组实现。用数组实现，就要处理扩容缩容的问题；用链表实现，没有这个问题，但需要更多的内存空间存储节点指针。

「图」的两种表示方法，邻接表就是链表，邻接矩阵就是二维数组。邻接矩阵判断连通性迅速，并可以进行矩阵运算解决一些问题，但是如果图比较稀疏的话很耗费空间。邻接表比较节省空间，但是很多操作的效率上肯定比不过邻接矩阵。

「散列表」就是通过散列函数把键映射到一个大数组里。而且对于解决散列冲突的方法，拉链法需要链表特性，操作简单，但需要额外的空间存储指针；线性探查法就需要数组特性，以便连续寻址，不需要指针的存储空间，但操作稍微复杂些。

「树」，用数组实现就是「堆」，因为「堆」是一个完全二叉树，用数组存储不需要节点指针，操作也比较简单；用链表实现就是很常见的那种「树」，因为不一定是完全二叉树，所以不适合用数组存储。为此，在这种链表「树」结构之上，又衍生出各种巧妙的设计，比如二叉搜索树、AVL 树、红黑树、区间树、B 树等等，以应对不同的问题。

了解 Redis 数据库的朋友可能也知道，Redis 提供列表、字符串、集合等等几种常用数据结构，但是对于每种数据结构，底层的存储方式都至少有两种，以便于根据存储数据的实际情况使用合适的存储方式。

综上，数据结构种类很多，甚至你也可以发明自己的数据结构，但是底层存储无非数组或者链表，**二者的优缺点如下**：

数组由于是紧凑连续存储,可以随机访问，通过索引快速找到对应元素，而且相对节约存储空间。但正因为连续存储，内存空间必须一次性分配够，所以说数组如果要扩容，需要重新分配一块更大的空间，再把数据全部复制过去，时间复杂度 O(N)；而且你如果想在数组中间进行插入和删除，每次必须搬移后面的所有数据以保持连续，时间复杂度 O(N)。

链表因为元素不连续，而是靠指针指向下一个元素的位置，所以不存在数组的扩容问题；如果知道某一元素的前驱和后驱，操作指针即可删除该元素或者插入新元素，时间复杂度 O(1)。但是正因为存储空间不连续，你无法根据一个索引算出对应元素的地址，所以不能随机访问；而且由于每个元素必须存储指向前后元素位置的指针，会消耗相对更多的储存空间。

二、数据结构的基本操作

对于任何数据结构，其基本操作无非遍历 + 访问，再具体一点就是：增删查改。

数据结构种类很多，但它们存在的目的都是在不同的应用场景，尽可能高效地增删查改。话说这不就是数据结构的使命么？

如何遍历 + 访问？我们仍然从最高层来看，各种数据结构的遍历 + 访问无非两种形式：线性的和非线性的。

线性就是 for/while 迭代为代表，非线性就是递归为代表。再具体一步，无非以下几种框架：

数组遍历框架，典型的线性迭代结构：

```
void traverse(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        // 迭代访问 arr[i]
    }
}
```

链表遍历框架，兼具迭代和递归结构：

```
/* 基本的单链表节点 */
class ListNode {
    int val;
    ListNode next;
}

void traverse(ListNode head) {
    for (ListNode p = head; p != null; p = p.next) {
        // 迭代访问 p.val
    }
}

void traverse(ListNode head) {
```

```
    // 递归访问 head.val
    traverse(head.next);
}
```

二叉树遍历框架，典型的非线性递归遍历结构：

```
/* 基本的二叉树节点 */
class TreeNode {
    int val;
    TreeNode left, right;
}

void traverse(TreeNode root) {
    traverse(root.left);
    traverse(root.right);
}
```

你看二叉树的递归遍历方式和链表的递归遍历方式，相似不？再看看二叉树结构和单链表结构，相似不？如果再多几条叉，N 叉树你会不会遍历？

二叉树框架可以扩展为 N 叉树的遍历框架：

```
/* 基本的 N 叉树节点 */
class TreeNode {
    int val;
    TreeNode[] children;
}

void traverse(TreeNode root) {
    for (TreeNode child : root.children)
        traverse(child);
}
```

`N` 叉树的遍历又可以扩展为图的遍历，因为图就是好几 `N` 叉棵树的结合体。你说图是可能出现环的？这个很好办，用个布尔数组 `visited` 做标记就行了，这里就不写代码了。

所谓框架，就是套路。不管增删查改，这些代码都是永远无法脱离的结构，你可以把这个结构作为大纲，根据具体问题在框架上添加代码就行了，下面会具体举例。

三、算法刷题指南

首先要明确的是，**数据结构是工具，算法是通过合适的工具解决特定问题的方法**。也就是说，学习算法之前，最起码得了解那些常用的数据结构，了解它们的特性和缺陷。

那么该如何在 LeetCode 刷题呢？之前的文章写过一些，什么按标签刷，坚持下去云云。现在距那篇文章已经过去将近一年了，我不说那些不痛不痒的话，直接说具体的建议：

先刷二叉树，先刷二叉树，先刷二叉树！

这是我这刷题一年的亲身体会，下图是去年十月份的提交截图：



公众号文章的阅读数据显示，大部分人对数据结构相关的算法文章不感兴趣，而是更关心动规回溯分治等等技巧。为什么要先刷二叉树呢，**因为二叉树是最容易培养框架思维的，而且大部分算法技巧，本质上都是树的遍历问题**。

刷二叉树看到题目没思路？根据很多读者的问题，其实大家不是没思路，只是没有理解我们说的「框架」是什么。

不要小看这几行破代码，几乎所有二叉树的题目都是一套这个框架就出来了：

```
void traverse(TreeNode root) {
    // 前序遍历代码位置
    traverse(root.left);
    // 中序遍历代码位置
    traverse(root.right);
    // 后序遍历代码位置
}
```

比如说我随便拿几道题的解法出来，不用管具体的代码逻辑，只要看看框架在其中是如何发挥作用的就行。

力扣第 124 题，难度困难，让你求二叉树中最大路径和，主要代码如下：

```
int res = Integer.MAX_VALUE;
int oneSideMax(TreeNode root) {
    if (root == null) return 0;
    int left = max(0, oneSideMax(root.left));
    int right = max(0, oneSideMax(root.right));
```

```
    // 后序遍历代码位置
    res = Math.max(res, left + right + root.val);
    return Math.max(left, right) + root.val;
}
```

注意递归函数的位置，这就是个后序遍历嘛，无非就是把 `traverse` 函数名字改成 `oneSideMax` 了。

力扣第 105 题，难度中等，让你根据前序遍历和中序遍历的结果还原一棵二叉树，很经典的问题吧，主要代码如下：

```
TreeNode build(int[] preorder, int preStart, int preEnd,
               int[] inorder, int inStart, int inEnd) {
    // 前序位置，寻找左右子树的索引
    if (preStart > preEnd) {
        return null;
    }
    int rootVal = preorder[preStart];
    int index = 0;
    for (int i = inStart; i <= inEnd; i++) {
        if (inorder[i] == rootVal) {
            index = i;
            break;
        }
    }
    int leftSize = index - inStart;
    TreeNode root = new TreeNode(rootVal);

    // 递归构造左右子树
    root.left = build(preorder, preStart + 1, preStart + leftSize,
                      inorder, inStart, index - 1);
    root.right = build(preorder, preStart + leftSize + 1, preEnd,
                       inorder, index + 1, inEnd);
    return root;
}
```

不要看这个函数的参数很多，只是为了控制数组索引而已。注意找递归函数的位置，本质上该算法也就是一个前序遍历，因为它在前序遍历的位置加了一坨代码。

力扣第 230 题，难度中等，寻找二叉搜索树中的第 `k` 大元素，主要代码如下：

```
int res = 0;
int rank = 0;
void traverse(TreeNode root, int k) {
    if (root == null) {
        return;
    }
    traverse(root.left, k);
    /* 中序遍历代码位置 */
    rank++;
```

```
        if (k == rank) {
            res = root.val;
            return;
        }
        /*****************/
        traverse(root.right, k);
    }
```

这不就是个中序遍历嘛，对于一棵 BST 中序遍历意味着什么，应该不需要解释了吧。

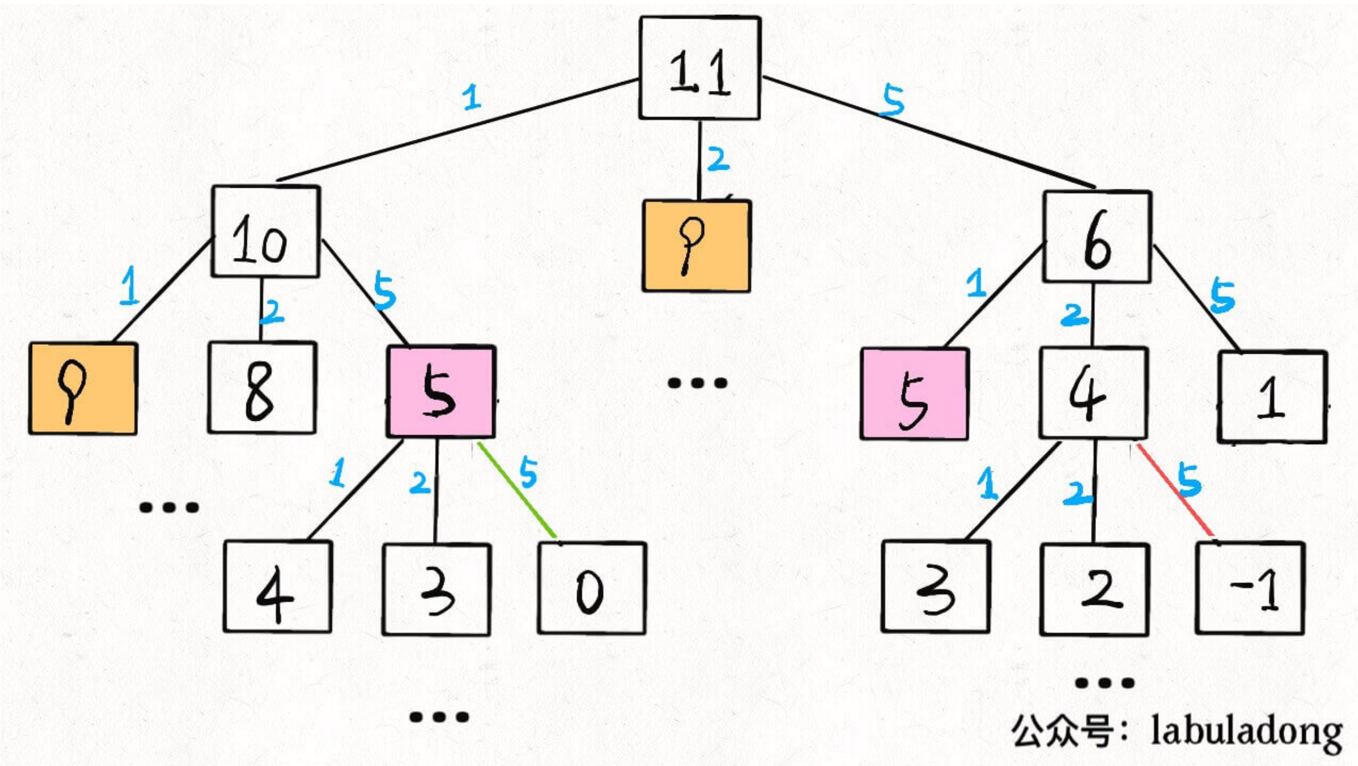
你看，二叉树的题目不过如此，只要把框架写出来，然后往相应的位置加代码就行了，这不就是思路吗。

对于一个理解二叉树的人来说，刷一道二叉树的题目花不了多长时间。那么如果你对刷题无从下手或者有畏惧心理，不妨从二叉树下手，前 10 道也许有点难受；结合框架再做 20 道，也许你就有点自己的理解了；刷完整个专题，再去做什么回溯动规分治专题，**你就会发现只要涉及递归的问题，都是树的问题**。

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

再举例吧，说几道我们之前文章写过的问题。

动态规划详解说过凑零钱问题，暴力解法就是遍历一棵 N 叉树：




```
int dp(int[] coins, int amount) {
    // base case
    if (amount == 0) return 0;
    if (amount < 0) return -1;

    int res = Integer.MAX_VALUE;
    for (int coin : coins) {
```

```
            int subProblem = dp(coins, amount - coin);
            // 子问题无解则跳过
            if (subProblem == -1) continue;
            // 在子问题中选择最优解，然后加一
            res = Math.min(res, subProblem + 1);
        }
        return res == Integer.MAX_VALUE ? -1 : res;
    }
```

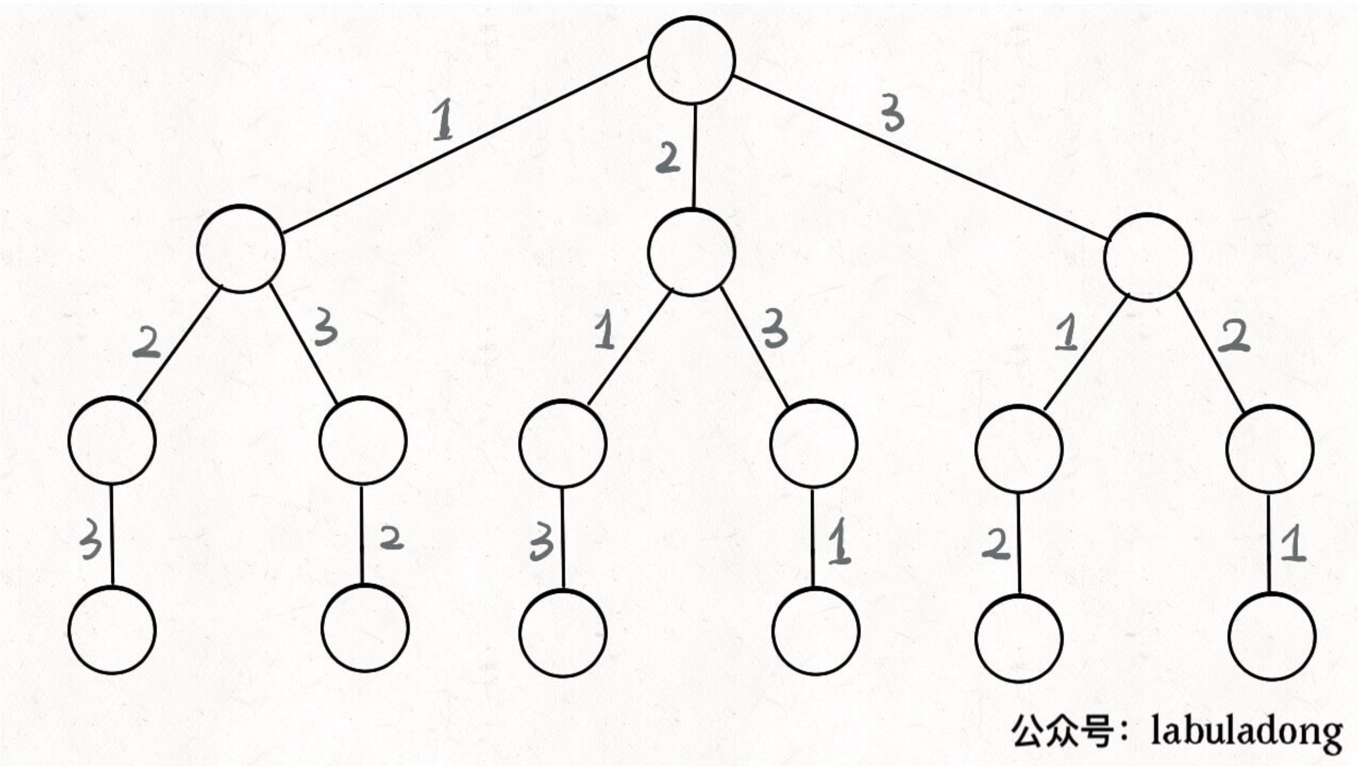
这么多代码看不懂咋办？直接提取出框架，就能看出核心思路了：

```
# 不过是一个 N 叉树的遍历问题而已
int dp(int amount) {
    for (int coin : coins) {
        dp(amount - coin);
    }
}
```

其实很多动态规划问题就是在遍历一棵树，你如果对树的遍历操作烂熟于心，起码知道怎么把思路转化成代码，也知道如何提取别人解法的核心思路。

再看看回溯算法，前文回溯算法详解干脆直接说了，回溯算法就是个 N 叉树的前后序遍历问题，没有例外。

比如全排列问题吧，本质上全排列就是在遍历下面这棵树，到叶子节点的路径就是一个全排列：



全排列算法的主要代码如下：

```
void backtrack(int[] nums, LinkedList<Integer> track) {
    if (track.size() == nums.length) {
        res.add(new LinkedList(track));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        if (track.contains(nums[i]))
            continue;
        track.add(nums[i]);
        // 进入下一层决策树
        backtrack(nums, track);
        track.removeLast();
    }

/* 提取出 N 叉树遍历框架 */
void backtrack(int[] nums, LinkedList<Integer> track) {
    for (int i = 0; i < nums.length; i++) {
        backtrack(nums, track);
}
```

N 叉树的遍历框架，找出来了吧？你说，树这种结构重不重要？

综上，对于畏惧算法的同学来说，可以先刷树的相关题目，试着从框架上看问题，而不要纠结于细节问题。

纠结细节问题，就比如纠结 `i` 到底应该加到 `n` 还是加到 `n - 1`，这个数组的大小到底应该开 `n` 还是 `n + 1`？

从框架上看问题，就是像我们这样基于框架进行抽取和扩展，既可以在看别人解法时快速理解核心逻辑，也有助于找到我们自己写解法时的思路方向。

当然，如果细节出错，你得不到正确的答案，但是只要有框架，你再错也错不到哪去，因为你的方向是对的。

但是，你要是心中没有框架，那么你根本无法解题，给了你答案，你也不会发现这就是个树的遍历问题。

这种思维是很重要的，动态规划详解中总结的找状态转移方程的几步流程，有时候按照流程写出解法，说实话我自己都不知道为啥是对的，反正它就是对了。。。

这就是框架的力量，能够保证你在快睡着的时候，依然能写出正确的程序；就算你啥都不会，都能比别人高一个级别。

四、总结几句

数据结构的基本存储方式就是链式和顺序两种，基本操作就是增删查改，遍历方式无非迭代和递归。

刷算法题建议从「树」分类开始刷，结合框架思维，把这几十道题刷完，对于树结构的理解应该就到位了。这时候去看回溯、动规、分治等算法专题，对思路的理解可能会更加深刻一些。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜

labuladong公众号

计算机算法的本质

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

两年前刚开这个公众号的时候，我写了一篇 学习数据结构和算法的框架思维，现在已经 5w 多阅读了，这对于一篇纯技术文来说是很牛逼的数据。

这两年在我自己不断刷题，思考和写公众号的过程中，我对算法的理解也是在逐渐加深，所以今天再写一篇，把我这两年的经验和思考浓缩成 4000 字，分享给大家。

本文主要有两部分，一是谈我对算法本质的理解，二是概括各种常用的算法。全文没有什么硬核的代码，都是我的经验之谈，也许没有多么高大上，但肯定能帮你少走弯路，更透彻地理解和掌握算法。

如果本文中你有不理解的地方大可跳过，多看一些我的历史文章之后再回过头看，大概就可以明白我想表达的意思了。

另外，本文包含大量历史文章链接，结合本文阅读历史文章也许可以更快培养出学习算法的框架思维和知识体系。

算法的本质

如果要让我一句话总结，我想说算法的本质就是「穷举」。

这么说肯定有人要反驳了，真的所有算法问题的本质都是穷举吗？没有一个例外吗？

例外肯定是有的，比如前几天我还发了 一行代码就能解决的算法题，这些题目都是通过观察，发现规律，然后找到最优解法。

再比如数学相关的算法，很多都是数学推论，然后用编程的形式表现出来了，所以它本质是数学，不是计算机算法。

从计算机算法的角度，结合我们大多数人的需求，这种秀智商的纯技巧题目绝对占少数，虽然很容易让人大呼精妙，但不能提炼出思考算法题的通用思维，真正通用的思维反而大道至简，就是穷举。

我记得自己一开始学习算法的时候，也觉得算法是一个很高大上的东西，每见到一道题，就想着能不能推导出一个什么数学公式，啪的一下就能把答案算出来。

比如你和一个没学过（计算机）算法的人说你写了个计算排列组合的算法，他大概以为你发明了一个公式，可以直接算出所有排列组合。但实际上呢？没什么高大上的公式，前文 回溯算法秒杀排列组合子集问题 写了，还是得用回溯算法暴力穷举。

对计算机算法的误解也许是以前学数学留下的「后遗症」，数学题一般都是你仔细观察，找几何关系，列方程，然后算出答案。如果说你需要进行大规模穷举来寻找答案，那大概率是你的解题思路出问题了。

而计算机解决问题的思维恰恰相反，有没有什么数学公式就交给你们人类去推导吧，如果能找到一些巧妙的定理那最好，但如果找不到，那就穷举呗，反正只要复杂度允许，没有什么答案是穷举不出来的。

技术岗笔试面试考的那些算法题，求个最大值最小值什么的，你怎么求？必须得把所有可行解穷举出来才能找到最值对吧，说白了不就这么点事儿么。

但是，你千万不要觉得穷举这个事儿很简单，穷举有两个关键难点：无遗漏、无冗余。

遗漏，会直接导致答案出错；冗余，会拖慢算法的运行速度。

所以，当你看到一道算法题，可以从这两个维度去思考：

1、如何穷举？即无遗漏地穷举所有可能解。

2、如何聪明地穷举？即避免所有冗余的计算。

不同类型的题目，难点是不同的，有的题目难在「如何穷举」，有的题目难在「如何聪明地穷举」。

什么算法的难点在「如何穷举」呢？一般是递归类问题，最典型的就是动态规划系列问题。

前文 动态规划核心套路 阐述了动态规划系列问题的核心原理，无非就是先写出暴力穷举解法（状态转移方程），加个备忘录就成自顶向下的递归解法了，再改一改就成自底向上的递推迭代解法了，动态规划的降维打击 里也讲过如何分析优化动态规划算法的空间复杂度。

上述过程就是在不断优化算法的时间、空间复杂度，也就是所谓「如何聪明地穷举」，这些技巧一听就会了。但很多读者留言说明白了这些原理，遇到动态规划题目还是不会做，因为第一步的暴力解法都写不出来。

这很正常，因为动态规划类型的题目可以千奇百怪，找状态转移方程才是难点，所以才有了 动态规划设计方法：数学归纳法 这篇文章，告诉你递归穷举的核心是数学归纳法，明确函数的定义，然后利用这个定义写递归函数，就可以穷举出所有可行解。

什么算法的难点在「如何聪明地穷举」呢？一些耳熟能详的非递归算法技巧，都可以归在这一类。

比如前文 Union Find 并查集算法详解 告诉你一种高效计算连通分量的技巧，理论上说，想判断两个节点是否连通，我用 DFS/BFS 暴力搜索（穷举）肯定可以做到，但人家 Union Find 算法硬是用数组模拟树结构，给你把连通性相关的操作复杂度给干到 `O(1)` 了。

这就属于聪明地穷举，你学过就会用，没学过恐怕很难想出这种思路。

再比如贪心算法技巧，前文 当老司机学会贪心算法 就告诉你，所谓贪心算法就是在题目中发现一些规律（专业点叫贪心选择性质），使得你不用完整穷举所有解就可以得出答案。

人家动态规划好歹是无冗余地穷举所有解，然后找一个最值，你贪心算法可好，都不用穷举所有解就可以找到答案，所以前文 贪心算法解决跳跃游戏 中贪心算法的效率比动态规划还高。

再比如大名鼎鼎的 KMP 算法，你写个字符串暴力匹配算法很容易，但你发明个 KMP 算法试试？KMP 算法的本质是聪明地缓存并复用一些信息，减少了冗余计算，前文 KMP 字符匹配算法 就是使用状态机的思路实现的 KMP 算法。

下面我概括性地列举一些常见的算法技巧，供大家学习参考。

数组/单链表系列算法

单链表常考的技巧就是双指针，前文 单链表六大技巧 全给你总结好了，这些技巧就是会者不难，难者不会。

比如判断单链表是否成环，拍脑袋的暴力解是什么？就是用一个 `HashSet` 之类的数据结构来缓存走过的节点，遇到重复的就说明有环对吧。但我们用快慢指针可以避免使用额外的空间，这就是聪明地穷举嘛。

当然，对于找链表中点这种问题，使用双指针技巧只是显示你学过这个技巧，和遍历两次链表的常规解法从时间空间复杂度的角度来说都是差不多的。

数组常用的技巧有很大一部分还是双指针相关的技巧，说白了是教你如何聪明地进行穷举。

首先说二分搜索技巧，可以归为两端向中心的双指针。如果让你在数组中搜索元素，一个 for 循环穷举肯定能搞定对吧，但如果数组是有序的，二分搜索不就是一种更聪明的搜索方式么。

前文 二分搜索框架详解 给你总结了二分搜索代码模板，保证不会出现搜索边界的问题。前文 二分搜索算法运用 给你总结了二分搜索相关题目的共性以及如何将二分搜索思想运用到实际算法中。

类似的两端向中心的双指针技巧还有力扣上的 N 数之和系列问题，前文 一个函数秒杀所有 nSum 问题 讲了这些题目的共性，甭管几数之和，解法肯定要穷举所有的数字组合，然后看看那个数字组合的和等于目标和嘛。比较聪明的方式是先排序，利用双指针技巧快速计算结果。

再说说 滑动窗口算法技巧，典型的快慢双指针，快慢指针中间就是滑动的「窗口」，主要用于解决子串问题。

文中最小覆盖子串这道题，让你寻找包含特定字符的最短子串，常规拍脑袋解法是什么？那肯定是类似字符串暴力匹配算法，用嵌套 for 循环穷举呗，平方级的复杂度。

而滑动窗口技巧告诉你不用这么麻烦，可以用快慢指针遍历一次就求出答案，这就是教你聪明的穷举技巧。

但是，就好像二分搜索只能运用在有序数组上一样，滑动窗口也是有其限制的，就是你必须明确的知道什么时候应该扩大窗口，什么时候该收缩窗口。

比如前文 最大子数组问题 面对的问题就没办法用滑动窗口，因为数组中的元素存在负数，扩大或缩小窗口并不能保证窗口中的元素之和就会随着增大和减小，所以无法使用滑动窗口技巧，只能用动态规划技巧穷举了。

还有回文串相关技巧，如果判断一个串是否是回文串，使用双指针从两端向中心检查，如果寻找回文子串，就从中心向两端扩散。前文 最长回文子串 使用了一种技巧同时处理了回文串长度为奇数或偶数的情况。

当然，寻找最长回文子串可以有更精妙的马拉车算法（Manacher 算法），不过，学习这个算法的性价比不高，没什么必要掌握。

最后说说 前缀和技巧 和 差分数组技巧。

如果频繁地让你计算子数组的和，每次用 for 循环去遍历肯定没问题，但前缀和技巧预计算一个 `preSum` 数组，就可以避免循环。

类似的，如果频繁地让你对子数组进行增减操作，也可以每次用 for 循环去操作，但差分数组技巧维护一个 `diff` 数组，也可以避免循环。

数组链表的技巧差不多就这些了，都比较固定，只要你都见过，运用出来的难度不算大，下面来说一说稍微有些难度的算法。

二叉树系列算法

老读者都知道，二叉树的重要性我之前说了无数次，因为二叉树模型几乎是所有高级算法的基础，尤其是那么多人说对递归的理解不到位，更应该好好刷二叉树相关题目。

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

我之前说过，二叉树题目的递归解法可以分两类思路，第一类是遍历一遍二叉树得出答案，第二类是通过分解问题计算出答案，这两类思路分别对应着 回溯算法核心框架 和 动态规划核心框架。

什么叫通过遍历一遍二叉树得出答案？

就比如说计算二叉树最大深度这个问题让你实现 `maxDepth` 这个函数，你这样写代码完全没问题：

```
// 记录最大深度
int res = 0;
int depth = 0;

// 主函数
int maxDepth(TreeNode root) {
    traverse(root);
    return res;
}

// 二叉树遍历框架
void traverse(TreeNode root) {
    if (root == null) {
        // 到达叶子节点
        res = Math.max(res, depth);
        return;
    }
    // 前序遍历位置
    depth++;
    traverse(root.left);
    traverse(root.right);
    // 后序遍历位置
    depth--;
}
```

这个逻辑就是用 `traverse` 函数遍历了一遍二叉树的所有节点，维护 `depth` 变量，在叶子节点的时候更新最大深度。

你看这段代码，有没有觉得很熟悉？能不能和回溯算法的代码模板对应上？

不信你照着 回溯算法核心框架 中全排列问题的代码对比下：

```java
// 记录所有全排列
List<List<Integer>> res = new LinkedList<>();
LinkedList<Integer> track = new LinkedList<>();

/* 主函数，输入一组不重复的数字，返回它们的全排列 */
List<List<Integer>> permute(int[] nums) {
    backtrack(nums);
    return res;
}

// 回溯算法框架
void backtrack(int[] nums) {
    if (track.size() == nums.length) {
        // 穷举完一个全排列
        res.add(new LinkedList(track));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        if (track.contains(nums[i]))
            continue;
        // 前序遍历位置做选择
        track.add(nums[i]);
        backtrack(nums);
        // 后序遍历位置取消选择
        track.removeLast();
    }
}
```

前文讲回溯算法的时候就告诉你回溯算法本质就是遍历一棵多叉树，连代码实现都如出一辙有没有？

而且我之前经常说，回溯算法虽然简单粗暴效率低，但特别有用，因为如果你对一道题无计可施，回溯算法起码能帮你写一个暴力解捞点分对吧。

那什么叫通过分解问题计算答案？

同样是计算二叉树最大深度这个问题，你也可以写出下面这样的解法：

```java
// 定义：输入根节点，返回这棵二叉树的最大深度
int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    // 递归计算左右子树的最大深度
    int leftMax = maxDepth(root.left);
    int rightMax = maxDepth(root.right);
    // 整棵树的最大深度
    int res = Math.max(leftMax, rightMax) + 1;

    return res;
}
```

你看这段代码，有没有觉得很熟悉？有没有觉得有点动态规划解法代码的形式？

不信你看 动态规划核心框架 中凑零钱问题的暴力穷举解法：

```
// 定义：输入金额 amount，返回凑出 amount 的最少硬币个数
int coinChange(int[] coins, int amount) {
    // base case
    if (amount == 0) return 0;
    if (amount < 0) return -1;

    int res = Integer.MAX_VALUE;
    for (int coin : coins) {
        // 递归计算凑出 amount - coin 的最少硬币个数
        int subProblem = coinChange(coins, amount - coin);
        if (subProblem == -1) continue;
        // 凑出 amount 的最少硬币个数
        res = Math.min(res, subProblem + 1);
    }

    return res == Integer.MAX_VALUE ? -1 : res;
}
```

这个暴力解法加个 memo 备忘录就是自顶向下的动态规划解法，你对照二叉树最大深度的解法代码，有没有发现很像？

如果你感受到最大深度这个问题两种解法的区别，那就趁热打铁，我问你，二叉树的前序遍历怎么写？

我相信大家都会对这个问题嗤之以鼻，毫不犹豫就可以写出下面这段代码：

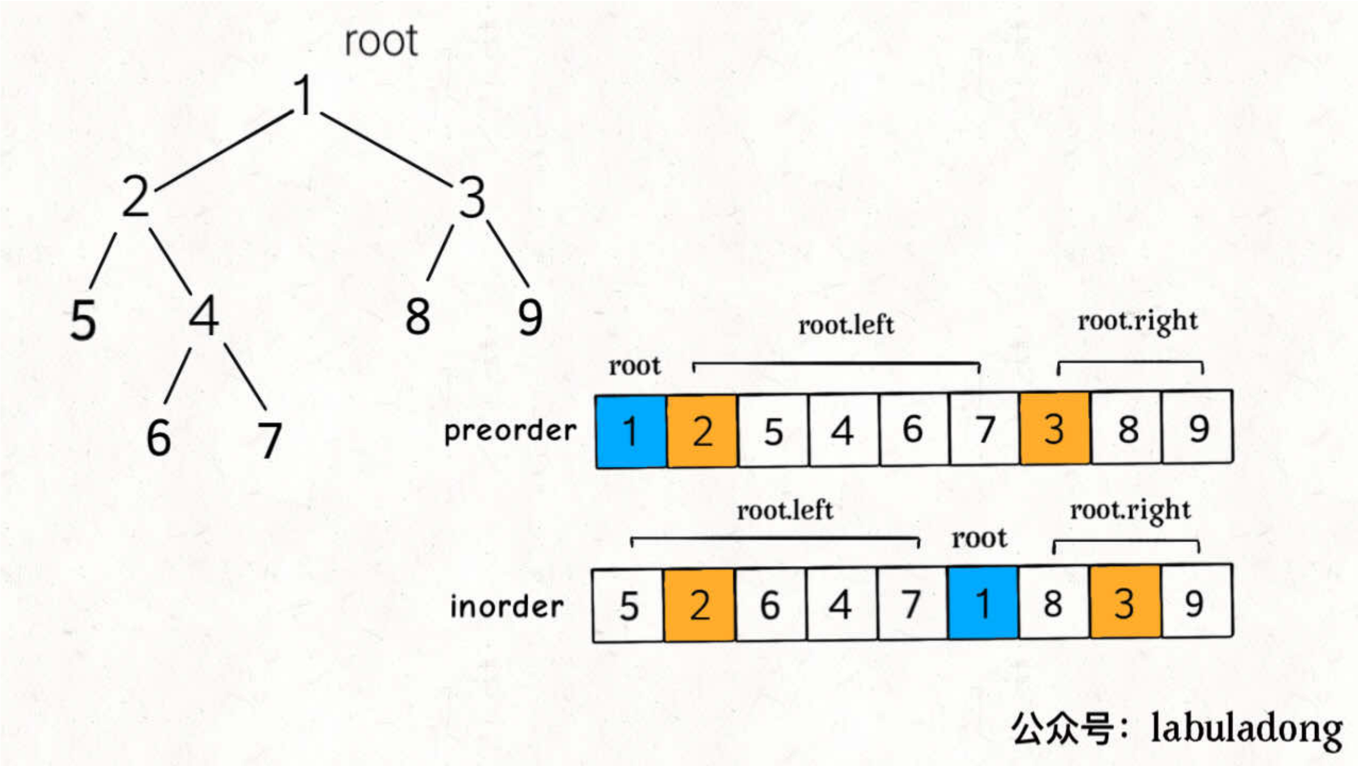
```
List<Integer> res = new LinkedList<>();

// 返回前序遍历结果
List<Integer> preorder(TreeNode root) {
    traverse(root);
    return res;
}

// 二叉树遍历函数
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序遍历位置
    res.add(root.val);
    traverse(root.left);
    traverse(root.right);
}
```

但是，你结合上面说到的两种不同的思维模式，二叉树的遍历是否也可以通过分解问题的思路解决呢？

我们前文 手把手刷二叉树（第二期） 说过前中后序遍历结果的特点：



你注意前序遍历的结果，根节点的值在第一位，后面接着左子树的前序遍历结果，最后接着右子树的前序遍历结果。

有没有体会出点什么来？其实完全可以重写前序遍历代码，用分解问题的形式写出来，避免外部变量和辅助函数：

```
// 定义：输入一棵二叉树的根节点，返回这棵树的前序遍历结果
List<Integer> preorder(TreeNode root) {
    List<Integer> res = new LinkedList<>();
    if (root == null) {
        return res;
    }
    // 前序遍历的结果，root.val 在第一个
    res.add(root.val);
    // 后面接着左子树的前序遍历结果
    res.addAll(preorder(root.left));
    // 最后接着右子树的前序遍历结果
    res.addAll(preorder(root.right));
}
```

你看，这就是用分解问题的思维模式写二叉树的前序遍历，如果写中序和后序遍历也是类似的。

当然，动态规划系列问题有「最优子结构」和「重叠子问题」两个特性，而且大多是让你求最值的。很多算法虽然不属于动态规划，但也符合分解问题的思维模式。

比如 分治算法详解 中说到的运算表达式优先级问题，其核心依然是大问题分解成子问题，只不过没有重叠子问题，不能用备忘录去优化效率罢了。

当然，除了动归、回溯（DFS）、分治，还有一个常用算法就是 BFS 了，前文 BFS 算法核心框架 就是根据下面这段二叉树的层序遍历代码改装出来的：

```
// 输入一棵二叉树的根节点，层序遍历这棵二叉树
void levelTraverse(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    int depth = 1;
    // 从上到下遍历二叉树的每一层
    while (!q.isEmpty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();

            if (cur.left != null) {
                q.offer(cur.left);
            }
            if (cur.right != null) {
                q.offer(cur.right);
            }
        }
        depth++;
    }
}
```

更进一步，图论相关的算法也是二叉树算法的延续。

比如 图论基础 和 环判断和拓扑排序 就用到了 DFS 算法；再比如 Dijkstra 算法模板，就是改造版 BFS 算法加上一个类似 dp table 的数组。

好了，说的差不多了，上述这些算法的本质都是穷举二（多）叉树，有机会的话通过剪枝或者备忘录的方式减少冗余计算，提高效率，就这么点事儿。

最后总结

上周在视频号直播的时候，有读者问我什么刷题方式是正确的，我说正确的刷题方式应该是刷一道题能获得刷十道题的效果，不然力扣现在 2000 道题目，你都打算刷完么？

那么怎么做到呢？学习数据结构和算法的框架思维 说了，要有框架思维，学会提炼重点，一个算法技巧可以包装出一百道题，如果你能一眼看穿它的本质，那就没必要浪费时间刷了嘛。

同时，在做题的时候要思考，联想，进而培养举一反三的能力。

前文 Dijkstra 算法模板 并不是真的是让你去背代码模板，不然的话直接甩出来那一段代码不就行了，我从层序遍历讲到 BFS 讲到 Dijkstra，说这么多废话干什么？

说到底我还是希望爱思考的读者能培养出成体系的算法思维，最好能爱上算法，而不是单纯地看题解去做题，授人以鱼不如授人以渔嘛。

本文就到这里吧，**算法真的没啥难的，只要有心，谁都可以学好**。分享是一种美德，如果本文对你有启发，欢迎分享给需要的朋友。

如果你有高质量的问题，可以 [点击这里](#) 进入本文的讨论区。

————————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



学剑篇、基础数据结构



基础数据结构包括数组、链表、队列、栈等，因为它们都比较类似，而且操作过程中不怎么涉及递归，所以我把它们归为较基础的数据结构。

公众号标签：手把手刷数据结构

1.1 数组/链表

数组链表代表着计算机最基本的两种存储形式：顺序存储和链式存储，所以他俩可以算是最基本的数据结构。

数组链表的主要算法技巧是双指针，双指针又分为中间向两端扩散的双指针、两端向中间收缩的双指针、快慢指针。

此外，数组还有前缀和和差分数组也属于必知必会的算法技巧。

公众号标签：链表双指针，数组双指针

小而美的算法技巧：前缀和数组

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

303. 区域和检索 - 数组不可变（中等）

304. 二维区域和检索 - 矩阵不可变（中等）

560. 和为K的子数组（中等）

前缀和技巧适用于快速、频繁地计算一个索引区间内的元素之和。

一维数组中的前缀和

先看一道例题，力扣第 303 题「区域和检索 - 数组不可变」，让你计算数组区间内元素的和，这是一道标准的前缀和问题：

303. 区域和检索 - 数组不可变

难度 简单 382

给定一个整数数组 `nums`，求出数组从索引 `i` 到 `j`（`i ≤ j`）范围内元素的总和，包含 `i`、`j` 两点。

实现 `NumArray` 类：

- `NumArray(int[] nums)` 使用数组 `nums` 初始化对象
- `int sumRange(int i, int j)` 返回数组 `nums` 从索引 `i` 到 `j`（`i ≤ j`）范围内元素的总和，包含 `i`、`j` 两点（也就是 `sum(nums[i], nums[i + 1], ... , nums[j])`）

示例：

```
输入：
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
输出：
[null, 1, -1, -3]

解释：
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return 1 ((-2) + 0 + 3)
numArray.sumRange(2, 5); // return -1 (3 + (-5) + 2 + (-1))
numArray.sumRange(0, 5); // return -3 ((-2) + 0 + 3 + (-5) + 2
+ (-1))
```

题目要求你实现这样一个类：

```java
class NumArray {

    public NumArray(int[] nums) {}

    /* 查询闭区间 [left, right] 的累加和 */
    public int sumRange(int left, int right) {}
}
```

`sumRange` 函数需要计算并返回一个索引区间之内的元素和，没学过前缀和的人可能写出如下代码：

```java
class NumArray {

    private int[] nums;

    public NumArray(int[] nums) {
        this.nums = nums;
    }

    public int sumRange(int left, int right) {
        int res = 0;
        for (int i = left; i <= right; i++) {
            res += nums[i];
        }
        return res;
    }
}
```

这样，可以达到效果，但是效率很差，因为 `sumRange` 方法会被频繁调用，而它的时间复杂度是 `O(N)`，其中 `N` 代表 `nums` 数组的长度。

这道题的最优解法是使用前缀和技巧，将 `sumRange` 函数的时间复杂度降为 `O(1)`，说白了就是不要在 `sumRange` 里面用 for 循环，咋整？

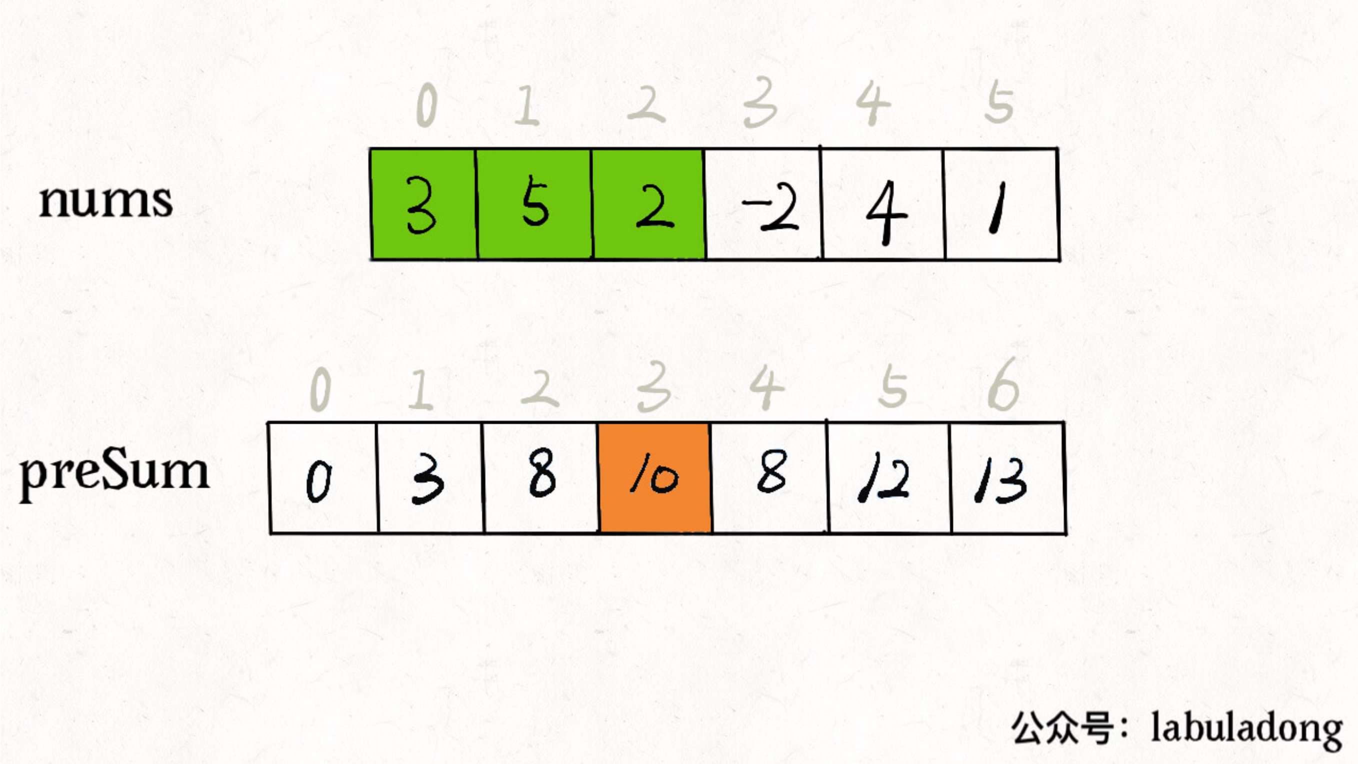
直接看代码实现：

```java
class NumArray {
    // 前缀和数组
    private int[] preSum;

    /* 输入一个数组，构造前缀和 */
    public NumArray(int[] nums) {
        // preSum[0] = 0，便于计算累加和
        preSum = new int[nums.length + 1];
        // 计算 nums 的累加和
        for (int i = 1; i < preSum.length; i++) {
            preSum[i] = preSum[i - 1] + nums[i - 1];
        }
    }

    /* 查询闭区间 [left, right] 的累加和 */
    public int sumRange(int left, int right) {
        return preSum[right + 1] - preSum[left];
    }
}
```

核心思路是我们 new 一个新的数组 `preSum` 出来，`preSum[i]` 记录 `nums[0..i-1]` 的累加和，看图 10 = 3 + 5 + 2：



看这个 preSum 数组，如果我想求索引区间 [1, 4] 内的所有元素之和，就可以通过 preSum[5] - preSum[1] 得出。

这样，sumRange 函数仅仅需要做一次减法运算，避免了每次进行 for 循环调用，最坏时间复杂度为常数 O(1)。

这个技巧在生活中运用也挺广泛的，比方说，你们班上有若干同学，每个同学有一个期末考试的成绩（满分 100 分），那么请你实现一个 API，输入任意一个分数段，返回有多少同学的成绩在这个分数段内。

那么，你可以先通过计数排序的方式计算每个分数具体有多少个同学，然后利用前缀和技巧来实现分数段查询的 API：

```
int[] scores; // 存储着所有同学的分数
// 试卷满分 100 分
int[] count = new int[100 + 1]
// 记录每个分数有几个同学
for (int score : scores)
    count[score]++
// 构造前缀和
for (int i = 1; i < count.length; i++)
    count[i] = count[i] + count[i-1];

// 利用 count 这个前缀和数组进行分数段查询
```

接下来，我们看一看前缀和思路在实际算法题中可以如何运用。

二维矩阵中的前缀和

这是力扣第 304 题「304. 二维区域和检索 - 矩阵不可变」，其实和上一题类似，上一题是让你计算子数组的元素之和，这道题让你计算二维矩阵中子矩阵的元素之和：

304. 二维区域和检索 - 矩阵不可变 labuladong 题解 思路

难度 中等 312

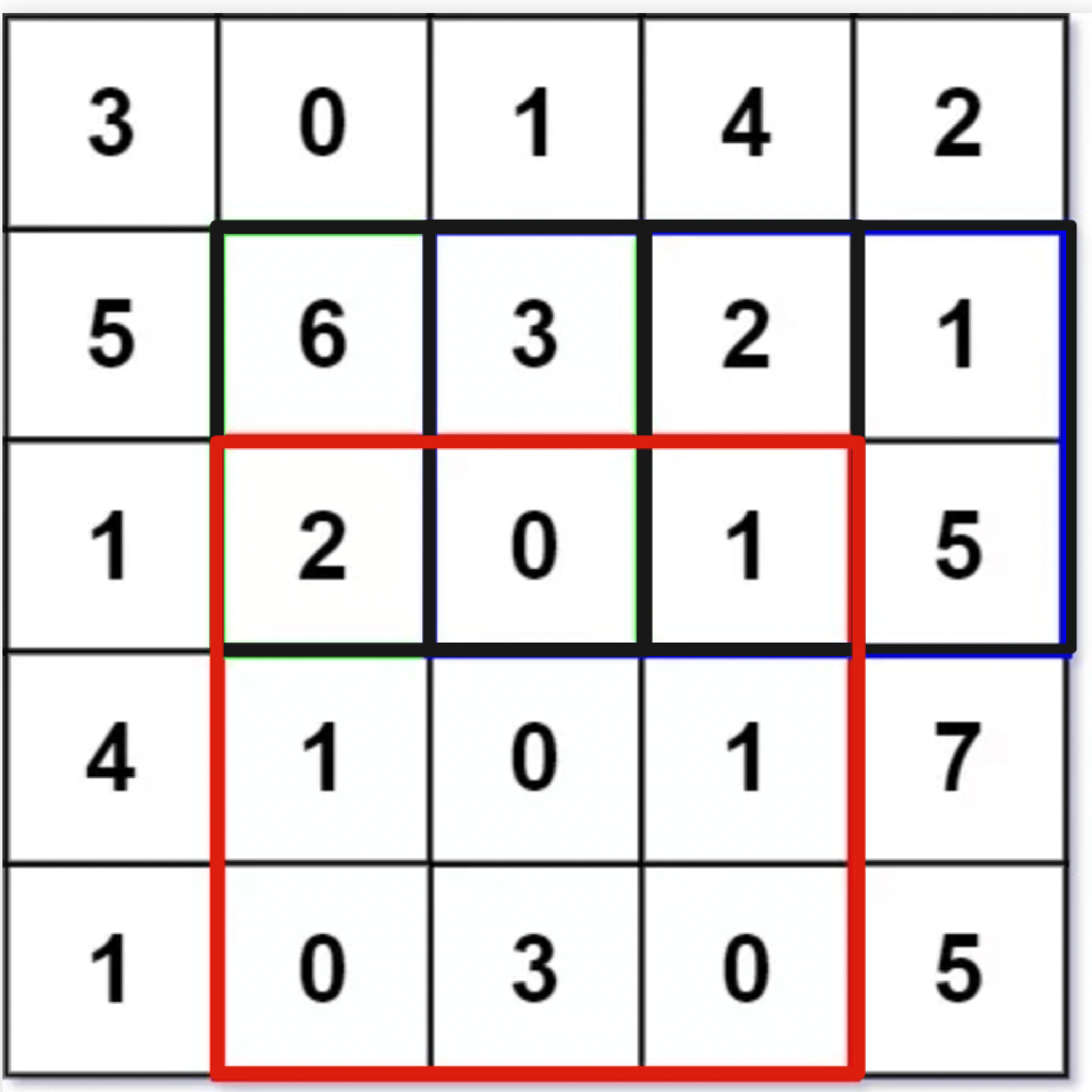
给定一个二维矩阵 `matrix`，以下类型的多个请求：

- 计算其子矩形范围内元素的总和，该子矩阵的 **左上角** 为 `(row1, col1)` ，**右下角** 为 `(row2, col2)` 。

实现 `NumMatrix` 类：

- `NumMatrix(int[][] matrix)` 给定整数矩阵 `matrix` 进行初始化
- `int sumRegion(int row1, int col1, int row2, int col2)` 返回 **左上角** `(row1, col1)` 、**右下角** `(row2, col2)` 所描述的子矩阵的元素 **总和** 。

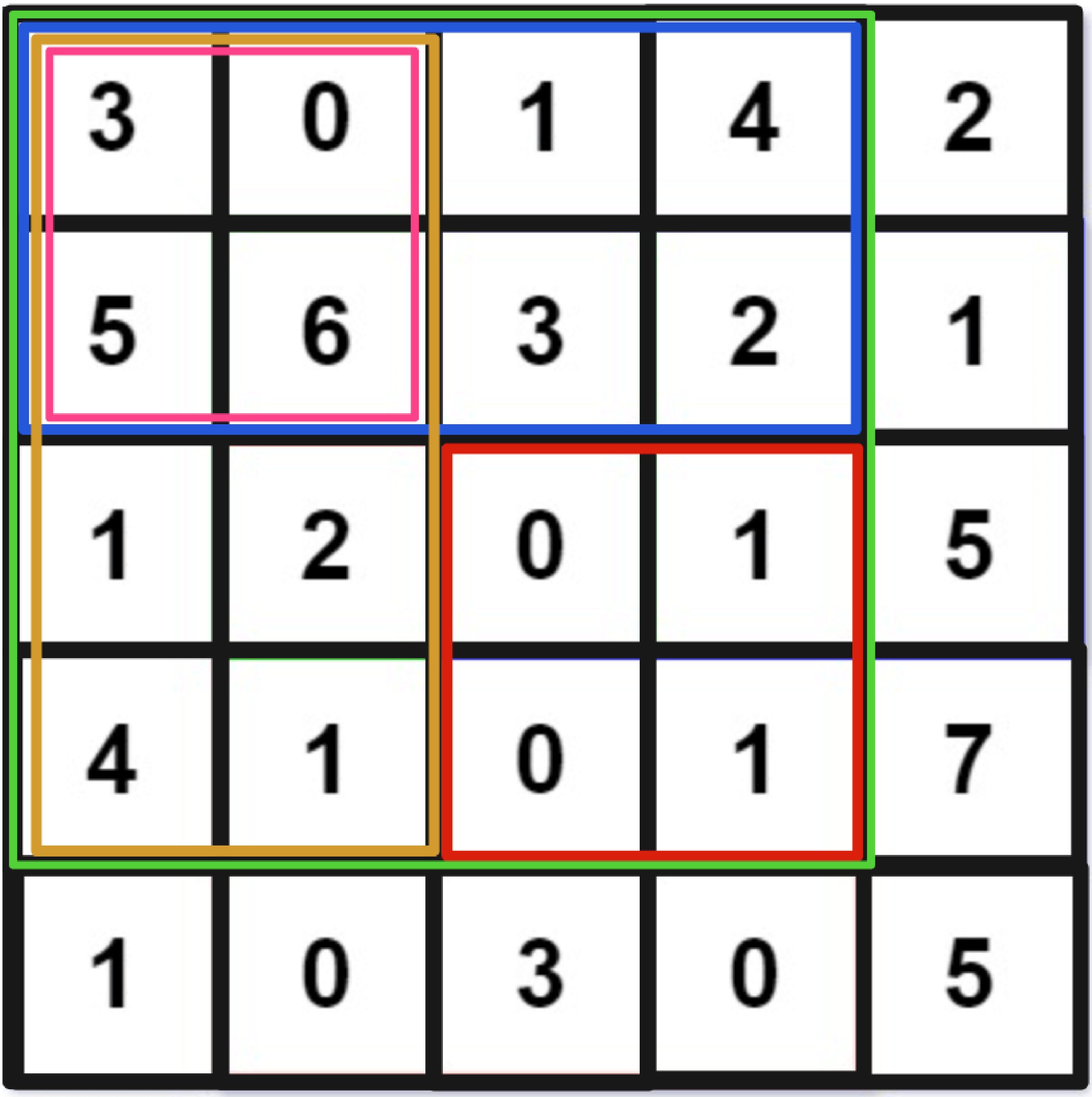
比如说输入的 `matrix` 如下图：



按照题目要求，矩阵左上角为坐标原点 `(0, 0)`，那么 `sumRegion([2,1,4,3])` 就是图中红色的子矩阵，你需要返回该子矩阵的元素和 8。

当然，你可以用一个嵌套 for 循环去遍历这个矩阵，但这样的话 `sumRegion` 函数的时间复杂度就高了，你算法的格局就低了。

做这道题更好的思路和一维数组中的前缀和是非常类似的，如下图：



如果我想计算红色的这个子矩阵的元素之和，可以用绿色矩阵减去蓝色矩阵减去橙色矩阵最后加上粉色矩阵，而绿蓝橙粉这四个矩阵有一个共同的特点，就是左上角就是 `(0, 0)` 原点。

那么我们可以维护一个二维 `preSum` 数组，专门记录以原点为顶点的矩阵的元素之和，就可以用几次加减运算算出任何一个子矩阵的元素和：

```
class NumMatrix {
    // 定义：preSum[i][j] 记录 matrix 中子矩阵 [0, 0, i-1, j-1] 的元素和
    private int[][] preSum;

    public NumMatrix(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
        if (m == 0 || n == 0) return;
        // 构造前缀和矩阵
        preSum = new int[m + 1][n + 1];
```

```
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // 计算每个矩阵 [0, 0, i, j] 的元素和
                preSum[i][j] = preSum[i-1][j] + preSum[i][j-1] + matrix[i
- 1][j - 1] - preSum[i-1][j-1];
            }
        }
    }

    // 计算子矩阵 [x1, y1, x2, y2] 的元素和
    public int sumRegion(int x1, int y1, int x2, int y2) {
        // 目标矩阵之和由四个相邻矩阵运算获得
        return preSum[x2+1][y2+1] - preSum[x1][y2+1] - preSum[x2+1][y1] +
preSum[x1][y1];
    }
}
```

这样，`sumRegion` 函数的时间复杂度也用前缀和技巧优化到了 O(1)，这是典型的「空间换时间」思路。

和为 k 的子数组

最后聊一道稍微有些困难的前缀和题目，力扣第 560 题「和为 K 的子数组」：

560. 和为 K 的子数组 labuladong 题解 思路

难度 中等 1154

给你一个整数数组 `nums` 和一个整数 `k`，请你统计并返回该数组中和为 `k` 的连续子数组的个数。

示例 1：

```
输入：nums = [1,1,1], k = 2
输出：2
```

示例 2：

```
输入：nums = [1,2,3], k = 3
输出：2
```

那我把所有子数组都穷举出来，算它们的和，看看谁的和等于 `k` 不就行了，借助前缀和技巧很容易写出一个解法：

```java
int subarraySum(int[] nums, int k) {
    int n = nums.length;
    // 构造前缀和
    int[] preSum = new int[n + 1];
    preSum[0] = 0;
    for (int i = 0; i < n; i++)
        preSum[i + 1] = preSum[i] + nums[i];

    int res = 0;
    // 穷举所有子数组
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < i; j++)
            // 子数组 nums[j..i-1] 的元素和
            if (preSum[i] - preSum[j] == k)
                res++;

    return res;
}
```

这个解法的时间复杂度 `O(N^2)` 空间复杂度 `O(N)`，并不是最优的解法。不过通过这个解法理解了前缀和数组的工作原理之后，可以使用一些巧妙的办法把时间复杂度进一步降低。

注意前面的解法有嵌套的 for 循环：

```java
for (int i = 1; i <= n; i++)
    for (int j = 0; j < i; j++)
        if (preSum[i] - preSum[j] == k)
            res++;
```

第二层 for 循环在干嘛呢？翻译一下就是，**在计算，有几个 `j` 能够使得 `preSum[i]` 和 `preSum[j]` 的差为 `k`**。每找到一个这样的 `j`，就把结果加一。

我们可以把 if 语句里的条件判断移项，这样写：

```java
if (preSum[j] == preSum[i] - k)
    res++;
```

优化的思路是：**我直接记录下有几个 `preSum[j]` 和 `preSum[i] - k` 相等，直接更新结果，就避免了内层的 for 循环**。我们可以用哈希表，在记录前缀和的同时记录该前缀和出现的次数。

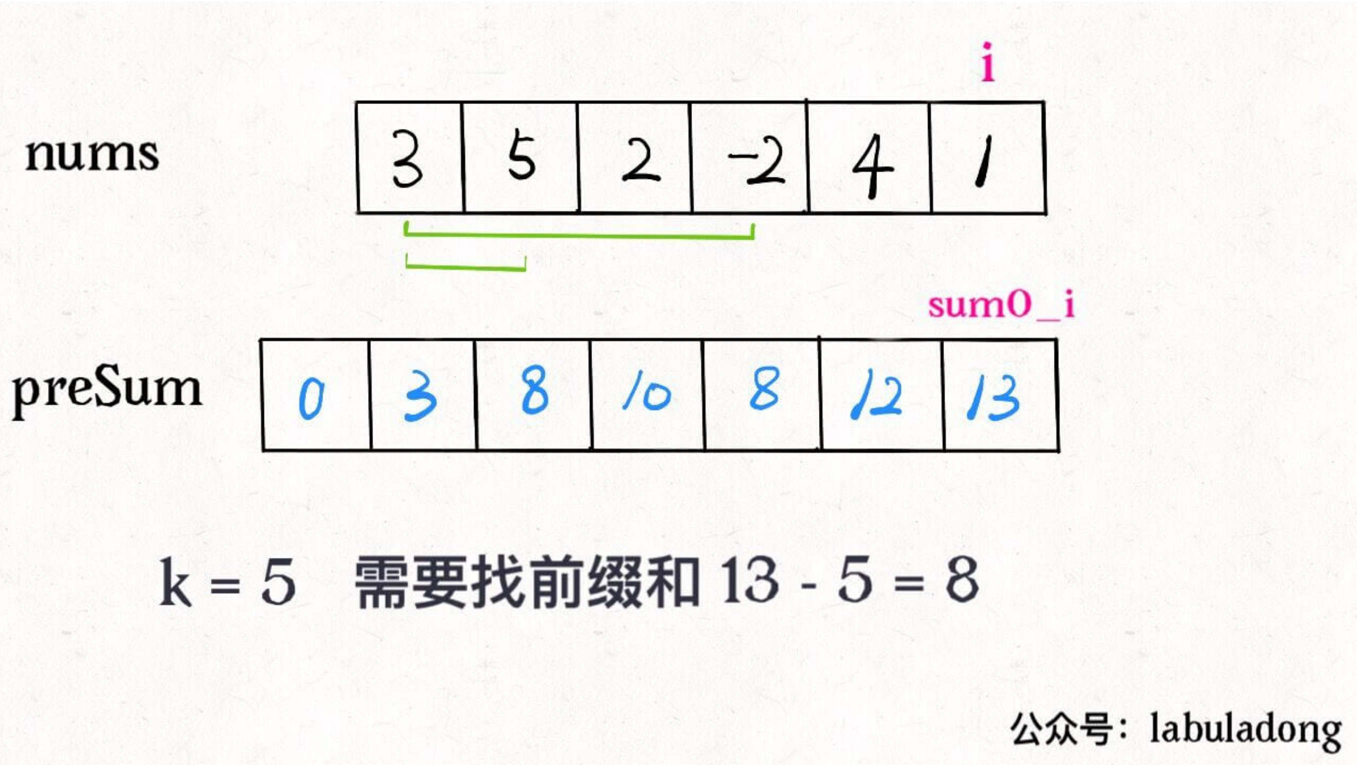
```java
int subarraySum(int[] nums, int k) {
    int n = nums.length;
    // map：前缀和 -> 该前缀和出现的次数
    HashMap<Integer, Integer>
        preSum = new HashMap<>();
    // base case
```

```java
        preSum.put(0, 1);

        int res = 0, sum0_i = 0;
        for (int i = 0; i < n; i++) {
            sum0_i += nums[i];
            // 这是我们想找的前缀和 nums[0..j]
            int sum0_j = sum0_i - k;
            // 如果前面有这个前缀和，则直接更新答案
            if (preSum.containsKey(sum0_j))
                res += preSum.get(sum0_j);
            // 把前缀和 nums[0..i] 加入并记录出现次数
            preSum.put(sum0_i,
                preSum.getOrDefault(sum0_i, 0) + 1);
        }
        return res;
    }
```

注意这里我们 `preSum` 记录的是前缀和到该前缀和出现的次数的映射。

比如说下面这个情况，需要前缀和 8 就能找到和为 `k` 的子数组了，之前的暴力解法需要遍历数组去数有几个 8，而优化解法借助哈希表可以直接得知有几个前缀和为 8。



这样，就把时间复杂度降到了 `O(N)`，是最优解法了。

前缀和技巧就讲到这里，应该说这个算法技巧是会者不难难者不会，实际运用中还是要多培养自己的思维灵活性，做到一眼看出题目是一个前缀和问题。

接下来可阅读：

- 差分数组技巧

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



小而美的算法技巧：差分数组

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

370. 区间加法（中等）

1109. 航班预订统计（中等）

1094. 拼车（中等）

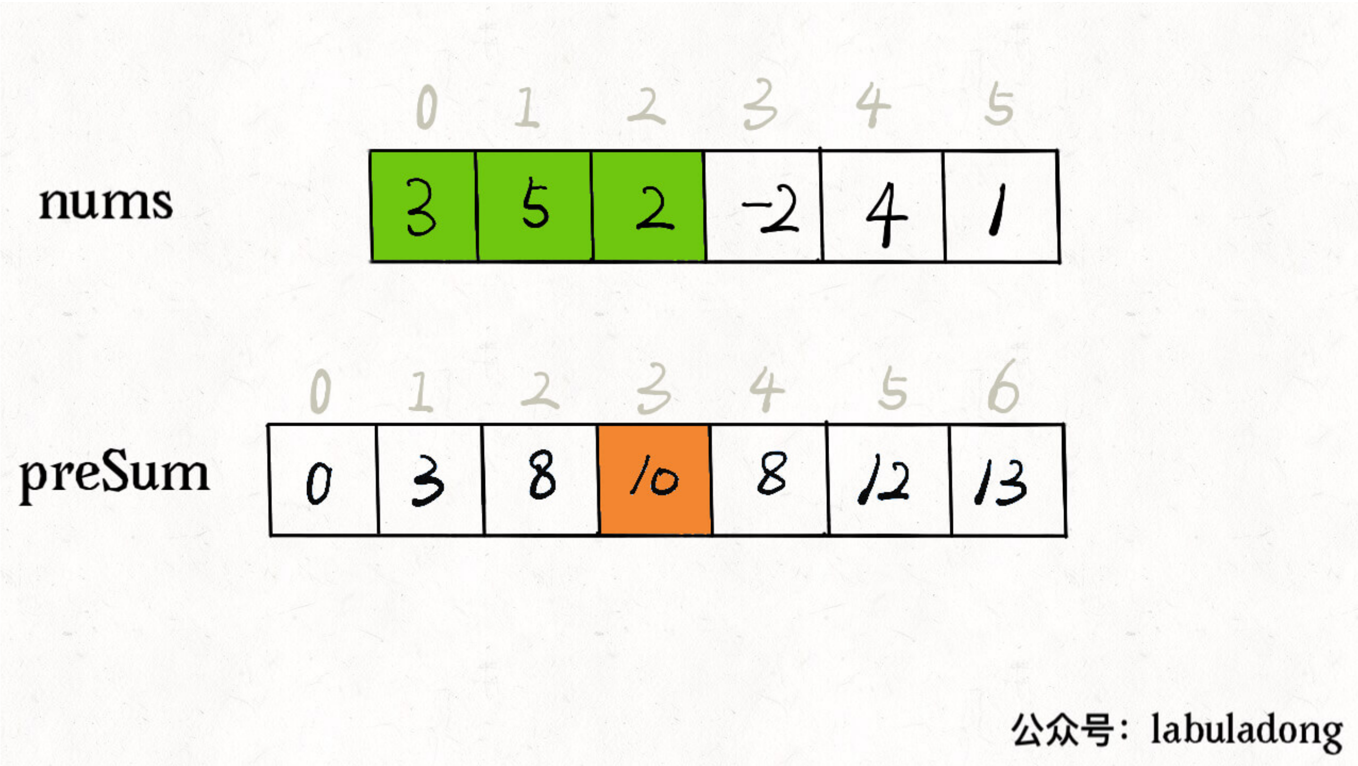
前文 前缀和技巧详解 写过的前缀和技巧是非常常用的算法技巧，**前缀和主要适用的场景是原始数组不会被修改的情况下，频繁查询某个区间的累加和。**

没看过前文没关系，这里简单介绍一下前缀和，核心代码就是下面这段：

```
class PrefixSum {
    // 前缀和数组
    private int[] prefix;

    /* 输入一个数组，构造前缀和 */
    public PrefixSum(int[] nums) {
        prefix = new int[nums.length + 1];
        // 计算 nums 的累加和
        for (int i = 1; i < prefix.length; i++) {
            prefix[i] = prefix[i - 1] + nums[i - 1];
        }
    }

    /* 查询闭区间 [i, j] 的累加和 */
    public int query(int i, int j) {
        return prefix[j + 1] - prefix[i];
    }
}
```



prefix[i] 就代表着 nums[0..i-1] 所有元素的累加和，如果我们想求区间 nums[i..j] 的累加和，只要计算 prefix[j+1] - prefix[i] 即可，而不需要遍历整个区间求和。

本文讲一个和前缀和思想非常类似的算法技巧「差分数组」，**差分数组的主要适用场景是频繁对原始数组的某个区间的元素进行增减**。

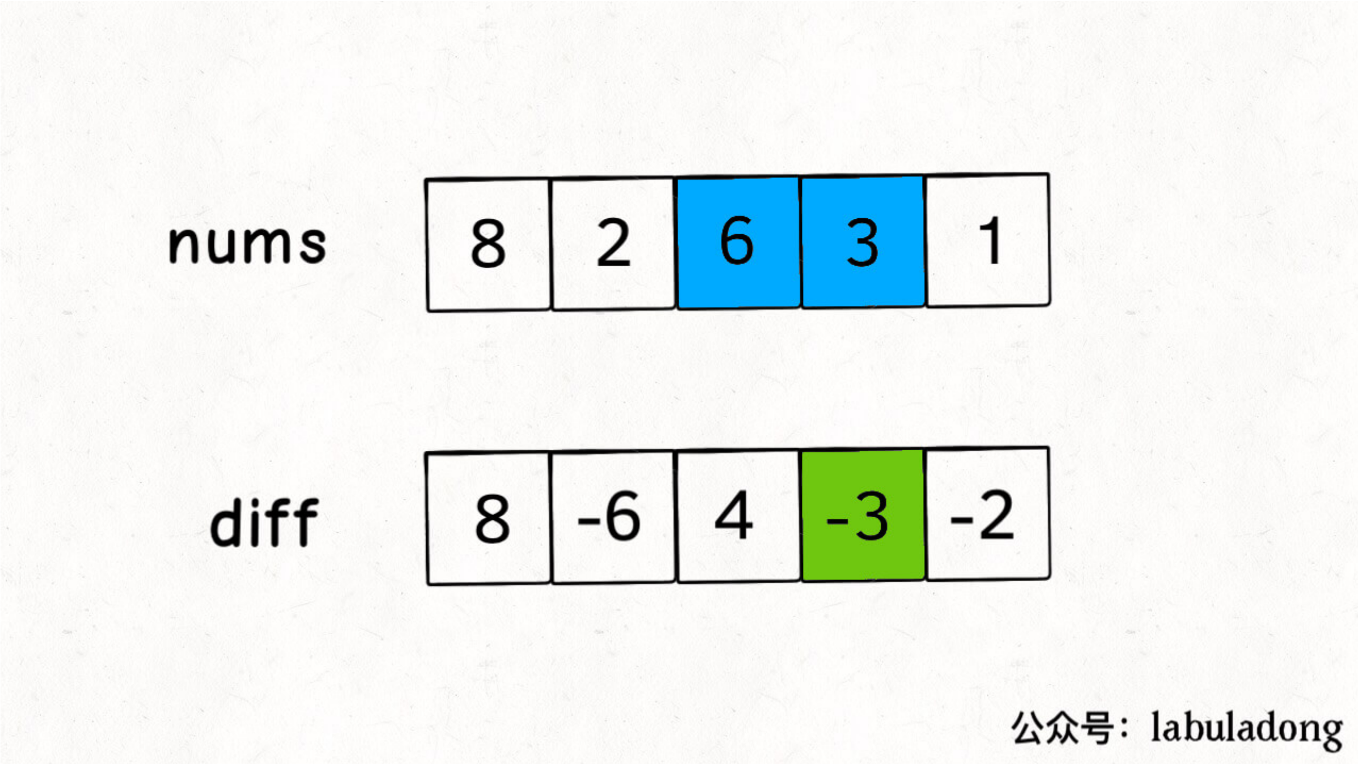
比如说，我给你输入一个数组 nums，然后又要求给区间 nums[2..6] 全部加 1，再给 nums[3..9] 全部减 3，再给 nums[0..4] 全部加 2，再给...

一通操作猛如虎，然后问你，最后 nums 数组的值是什么？

常规的思路很容易，你让我给区间 nums[i..j] 加上 val，那我就一个 for 循环给它们都加上呗，还能咋样？这种思路的时间复杂度是 O(N)，由于这个场景下对 nums 的修改非常频繁，所以效率会很低下。

这里就需要差分数组的技巧，类似前缀和技巧构造的 prefix 数组，我们先对 nums 数组构造一个 diff 差分数组，**diff[i] 就是 nums[i] 和 nums[i-1] 之差**：

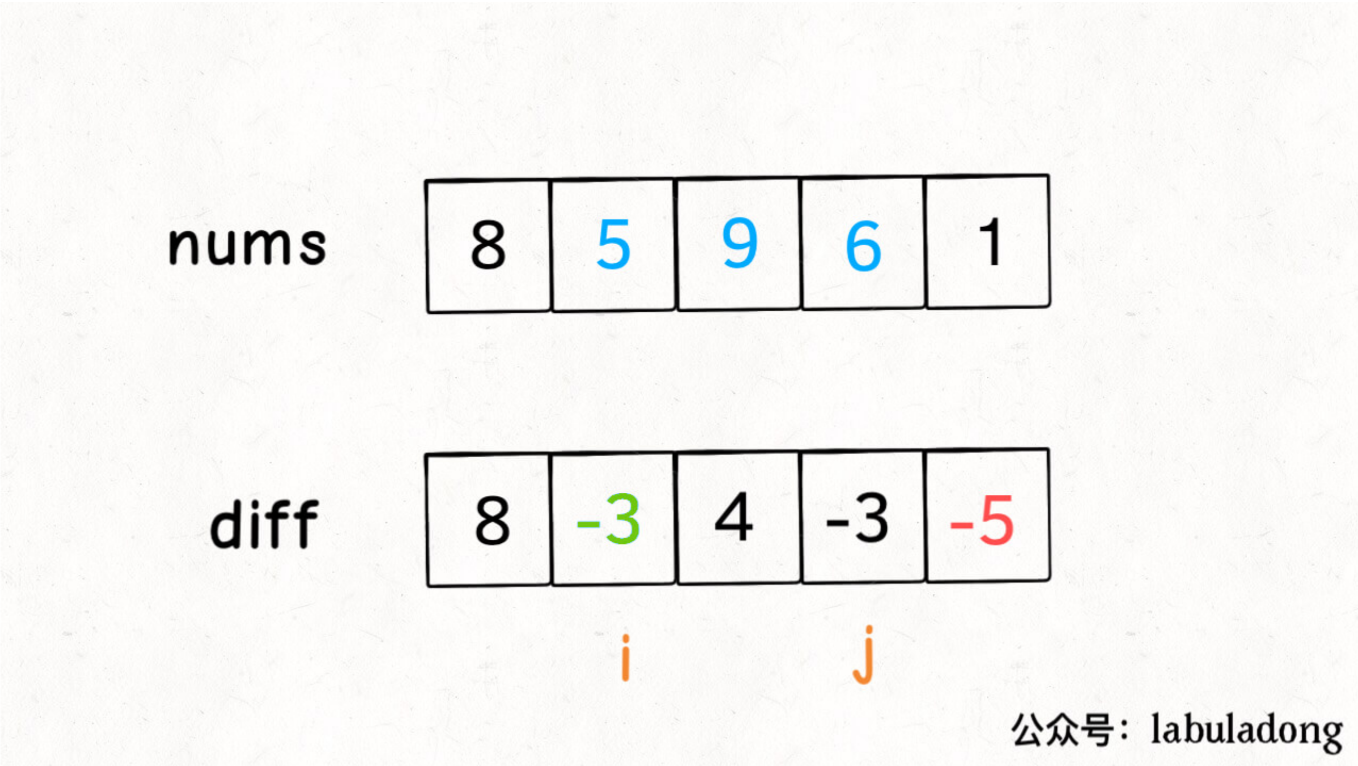
```
int[] diff = new int[nums.length];
// 构造差分数组
diff[0] = nums[0];
for (int i = 1; i < nums.length; i++) {
    diff[i] = nums[i] - nums[i - 1];
}
```



通过这个 diff 差分数组是可以反推出原始数组 nums 的，代码逻辑如下：

```
int[] res = new int[diff.length];
// 根据差分数组构造结果数组
res[0] = diff[0];
for (int i = 1; i < diff.length; i++) {
    res[i] = res[i - 1] + diff[i];
}
```

这样构造差分数组 diff，就可以快速进行区间增减的操作，如果你想对区间 nums[i..j] 的元素全部加 3，那么只需要让 diff[i] += 3，然后再让 diff[j+1] -= 3 即可：



原理很简单，回想 `diff` 数组反推 `nums` 数组的过程，`diff[i] += 3` 意味着给 `nums[i..]` 所有的元素都加了 3，然后 `diff[j+1] -= 3` 又意味着对于 `nums[j+1..]` 所有元素再减 3，那综合起来，是不是就是对 `nums[i..j]` 中的所有元素都加 3 了？

只要花费 O(1) 的时间修改 `diff` 数组，就相当于给 `nums` 的整个区间做了修改。多次修改 `diff`，然后通过 `diff` 数组反推，即可得到 `nums` 修改后的结果。

现在我们把差分数组抽象成一个类，包含 `increment` 方法和 `result` 方法：

```java
// 差分数组工具类
class Difference {
    // 差分数组
    private int[] diff;

    /* 输入一个初始数组，区间操作将在这个数组上进行 */
    public Difference(int[] nums) {
        assert nums.length > 0;
        diff = new int[nums.length];
        // 根据初始数组构造差分数组
        diff[0] = nums[0];
        for (int i = 1; i < nums.length; i++) {
            diff[i] = nums[i] - nums[i - 1];
        }
    }

    /* 给闭区间 [i,j] 增加 val（可以是负数）*/
    public void increment(int i, int j, int val) {
        diff[i] += val;
        if (j + 1 < diff.length) {
            diff[j + 1] -= val;
        }
```

```
    }

    /* 返回结果数组 */
    public int[] result() {
        int[] res = new int[diff.length];
        // 根据差分数组构造结果数组
        res[0] = diff[0];
        for (int i = 1; i < diff.length; i++) {
            res[i] = res[i - 1] + diff[i];
        }
        return res;
    }
}
```

这里注意一下 `increment` 方法中的 if 语句：

```
public void increment(int i, int j, int val) {
    diff[i] += val;
    if (j + 1 < diff.length) {
        diff[j + 1] -= val;
    }
}
```

当 `j+1 >= diff.length` 时，说明是对 `nums[i]` 及以后的整个数组都进行修改，那么就不需要再给 `diff` 数组减 `val` 了。

算法实践

首先，力扣第 370 题「区间加法」 就直接考察了差分数组技巧：

370. 区间加法 labuladong 题解 思路

难度 中等 81 收藏 分享 切换为英文 接收动态 反馈

假设你有一个长度为 ***n*** 的数组，初始情况下所有的数字均为 **0**，你将会被给出 ***k*** 个更新的操作。

其中，每个操作会被表示为一个三元组：**[startIndex, endIndex, inc]**，你需要将子数组 **A[startIndex ... endIndex]**（包括 startIndex 和 endIndex）增加 **inc**。

请你返回 ***k*** 次操作后的数组。

示例:

```
输入: length = 5, updates = [[1,3,2],[2,4,3],[0,2,-2]]
输出: [-2,0,3,5,3]
```

解释:

```
初始状态:
[0,0,0,0,0]

进行了操作 [1,3,2] 后的状态:
[0,2,2,2,0]

进行了操作 [2,4,3] 后的状态:
[0,2,5,5,3]

进行了操作 [0,2,-2] 后的状态:
[-2,0,3,5,3]
```

那么我们直接复用刚才实现的 `Difference` 类就能把这道题解决掉：

```
int[] getModifiedArray(int length, int[][] updates) {
    // nums 初始化为全 0
    int[] nums = new int[length];
    // 构造差分解法
    Difference df = new Difference(nums);

    for (int[] update : updates) {
        int i = update[0];
        int j = update[1];
        int val = update[2];
        df.increment(i, j, val);
    }
```

```
        return df.result();
    }
```

当然，实际的算法题可能需要我们对题目进行联想和抽象，不会这么直接地让你看出来要用差分数组技巧，这里看一下力扣第 1109 题「航班预订统计」：

1109. 航班预订统计 labuladong 题解 思路

难度 中等 84

这里有 `n` 个航班，它们分别从 `1` 到 `n` 进行编号。

我们这儿有一份航班预订表，表中第 `i` 条预订记录 `bookings[i] = [i, j, k]` 意味着我们在从 `i` 到 `j` 的每个航班上预订了 `k` 个座位。

请你返回一个长度为 `n` 的数组 `answer`，按航班编号顺序返回每个航班上预订的座位数。

示例：

```
输入: bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5
输出: [10,55,45,25,25]
```

函数签名如下：

```
int[] corpFlightBookings(int[][] bookings, int n)
```

这个题目就在那绕弯弯，其实它就是个差分数组的题，我给你翻译一下：

给你输入一个长度为 `n` 的数组 `nums`，其中所有元素都是 0。再给你输入一个 `bookings`，里面是若干三元组 `(i,j,k)`，每个三元组的含义就是要求你给 `nums` 数组的闭区间 `[i-1,j-1]` 中所有元素都加上 `k`。请你返回最后的 `nums` 数组是多少？

> PS：因为题目说的 `n` 是从 1 开始计数的，而数组索引从 0 开始，所以对于输入的三元组 `(i,j,k)`，数组区间应该对应 `[i-1,j-1]`。

这么一看，不就是一道标准的差分数组题嘛？我们可以直接复用刚才写的类：

```
int[] corpFlightBookings(int[][] bookings, int n) {
    // nums 初始化为全 0
    int[] nums = new int[n];
    // 构造差分解法
    Difference df = new Difference(nums);

    for (int[] booking : bookings) {
        // 注意转成数组索引要减一哦
```

```
            int i = booking[0] - 1;
            int j = booking[1] - 1;
            int val = booking[2];
            // 对区间 nums[i..j] 增加 val
            df.increment(i, j, val);
        }
        // 返回最终的结果数组
        return df.result();
    }
```

这道题就解决了。

还有一道很类似的题目是力扣第 1094 题「拼车」，我简单描述下题目：

你是一个开公交车的司机，公交车的最大载客量为 `capacity`，沿途要经过若干车站，给你一份乘客行程表 `int[][] trips`，其中 `trips[i] = [num, start, end]` 代表着有 `num` 个旅客要从站点 `start` 上车，到站点 `end` 下车，请你计算是否能够一次把所有旅客运送完毕（不能超过最大载客量 `capacity`）。

函数签名如下：

```
boolean carPooling(int[][] trips, int capacity);
```

比如输入：

```
trips = [[2,1,5],[3,3,7]], capacity = 4
```

这就不能一次运完，因为 `trips[1]` 最多只能上 2 人，否则车就会超载。

相信你已经能够联想到差分数组技巧了：**`trips[i]` 代表着一组区间操作，旅客的上车和下车就相当于数组的区间加减；只要结果数组中的元素都小于 `capacity`，就说明可以不超载运输所有旅客。**

但问题是，差分数组的长度（车站的个数）应该是多少呢？题目没有直接给，但给出了数据取值范围：

```
0 <= trips[i][1] < trips[i][2] <= 1000
```

车站个数最多为 1000，那么我们的差分数组长度可以直接设置为 1001：

```
boolean carPooling(int[][] trips, int capacity) {
    // 最多有 1000 个车站
    int[] nums = new int[1001];
    // 构造差分解法
    Difference df = new Difference(nums);

    for (int[] trip : trips) {
        // 乘客数量
```

```java
        int val = trip[0];
        // 第 trip[1] 站乘客上车
        int i = trip[1];
        // 第 trip[2] 站乘客已经下车,
        // 即乘客在车上的区间是 [trip[1], trip[2] - 1]
        int j = trip[2] - 1;
        // 进行区间操作
        df.increment(i, j, val);
    }

    int[] res = df.result();

    // 客车自始至终都不应该超载
    for (int i = 0; i < res.length; i++) {
        if (capacity < res[i]) {
            return false;
        }
    }
    return true;
}
```

至此，这道题也解决了。

最后，差分数组和前缀和数组都是比较常见且巧妙的算法技巧，分别适用不同的常见，而且是会者不难，难者不会。所以，关于差分数组的使用，你学会了吗？

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



我写了首诗，把滑动窗口算法算法变成了默写题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

76. 最小覆盖子串（困难）

567. 字符串的排列（中等）

438. 找到字符串中所有字母异位词（中等）

3. 无重复字符的最长子串（中等）

鉴于前文 二分搜索框架详解 的那首《二分搜索升天词》很受好评，并在民间广为流传，成为安睡助眠的一剂良方，今天在滑动窗口算法框架中，我再次编写一首小诗来歌颂滑动窗口算法的伟大：

滑动窗口防滑记
作者：labuladong

链表子串数组题，用双指针别犹豫。
双指针家三兄弟，各个都是万人迷。

快慢指针最神奇，链表操作无压力。
归并排序找中点，链表成环搞判定。

左右指针最常见，左右两端相向行。
反转数组要靠它，二分搜索是弟弟。

滑动窗口老猛男，子串问题全靠它。
左右指针滑窗口，一前一后齐头进。
自称十年老司机，怎料农村道路滑。
一不小心滑到了，鼻青脸肿少颗牙。
算法思想很简单，出了bug想升天。

labuladong稳若狗，一套框架不翻车。
一路漂移带闪电，算法变成默写题。
此车还有副驾驶，文末别忘瞧一瞧。

关于双指针的快慢指针和左右指针的用法，可以参见前文 双指针技巧汇总，本文就解决一类最难掌握的双指针技巧：滑动窗口技巧。总结出一套框架，可以保你闭着眼睛都能写出正确的解法。

说起滑动窗口算法，很多读者都会头疼。这个算法技巧的思路非常简单，就是维护一个窗口，不断滑动，然后更新答案么。LeetCode 上有起码 10 道运用滑动窗口算法的题目，难度都是中等和困难。该算法的大致逻辑如下：

```
int left = 0, right = 0;

while (right < s.size()) {
    // 增大窗口
    window.add(s[right]);
    right++;

    while (window needs shrink) {
```

```
            // 缩小窗口
            window.remove(s[left]);
            left++;
        }
    }
```

这个算法技巧的时间复杂度是 O(N)，比字符串暴力算法要高效得多。

其实困扰大家的，不是算法的思路，而是各种细节问题。比如说如何向窗口中添加新元素，如何缩小窗口，在窗口滑动的哪个阶段更新结果。即便你明白了这些细节，也容易出 bug，找 bug 还不知道怎么找，真的挺让人心烦的。

所以今天我就写一套滑动窗口算法的代码框架，我连再哪里做输出 debug 都给你写好了，以后遇到相关的问题，你就默写出来如下框架然后改三个地方就行，还不会出 bug：

```
/* 滑动窗口算法框架 */
void slidingWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...

        /*** debug 输出的位置 ***/
        printf("window: [%d, %d)\n", left, right);
        /********************/

        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
            // d 是将移出窗口的字符
            char d = s[left];
            // 左移窗口
            left++;
            // 进行窗口内数据的一系列更新
            ...
        }
    }
}
```

其中两处 `...` 表示的更新窗口数据的地方，到时候你直接往里面填就行了。

而且，这两个 `...` 处的操作分别是右移和左移窗口更新操作，等会你会发现它们操作是完全对称的。

说句题外话，我发现很多人喜欢执着于表象，不喜欢探求问题的本质。比如说有很多人评论我这个框架，说什么散列表速度慢，不如用数组代替散列表；还有很多人喜欢把代码写得特别短小，说我这样代码太多余，影响编译速度，LeetCode 上速度不够快。

我服了。算法看的是时间复杂度，你能确保自己的时间复杂度最优，就行了。至于 LeetCode 所谓的运行速度，那个都是玄学，只要不是慢的离谱就没啥问题，根本不值得你从编译层面优化，不要舍本逐末……

我的公众号重点在于算法思想，你把框架思维了然于心，然后随你魔改代码好吧，你高兴就好。

言归正传，下面就直接上**四道** LeetCode 原题来套这个框架，其中第一道题会详细说明其原理，后面四道就直接闭眼睛秒杀了。

因为滑动窗口很多时候都是在处理字符串相关的问题，Java 处理字符串不方便，所以本文代码为 C++ 实现。不会用到什么编程方面的奇技淫巧，但是还是简单介绍一下一些用到的数据结构，以免有的读者因为语言的细节问题阻碍对算法思想的理解：

`unordered_map` 就是哈希表（字典），它的一个方法 `count(key)` 相当于 Java 的 `containsKey(key)` 可以判断键 key 是否存在。

可以使用方括号访问键对应的值 `map[key]`。需要注意的是，如果该 `key` 不存在，C++ 会自动创建这个 key，并把 `map[key]` 赋值为 0。

所以代码中多次出现的 `map[key]++` 相当于 Java 的 `map.put(key, map.getOrDefault(key, 0) + 1)`。

一、最小覆盖子串

先来看看力扣第 76 题「最小覆盖子串」难度 Hard：

76. 最小覆盖子串 labuladong 题解 思路

难度 困难 1656

给你一个字符串 `s` 、一个字符串 `t` 。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""` 。

示例 1：

```
输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"
```

示例 2：

```
输入：s = "a", t = "a"
输出："a"
```

就是说要在 S(source) 中找到包含 T(target) 中全部字母的一个子串，且这个子串一定是所有可能子串中最短的。

如果我们使用暴力解法，代码大概是这样的：

```
for (int i = 0; i < s.size(); i++)
    for (int j = i + 1; j < s.size(); j++)
        if s[i:j] 包含 t 的所有字母:
            更新答案
```

思路很直接，但是显然，这个算法的复杂度肯定大于 O(N^2) 了，不好。

滑动窗口算法的思路是这样：

1、我们在字符串 S 中使用双指针中的左右指针技巧，初始化 `left = right = 0`，把索引**左闭右开**区间 `[left, right)` 称为一个「窗口」。

2、我们先不断地增加 `right` 指针扩大窗口 `[left, right)`，直到窗口中的字符串符合要求（包含了 T 中的所有字符）。

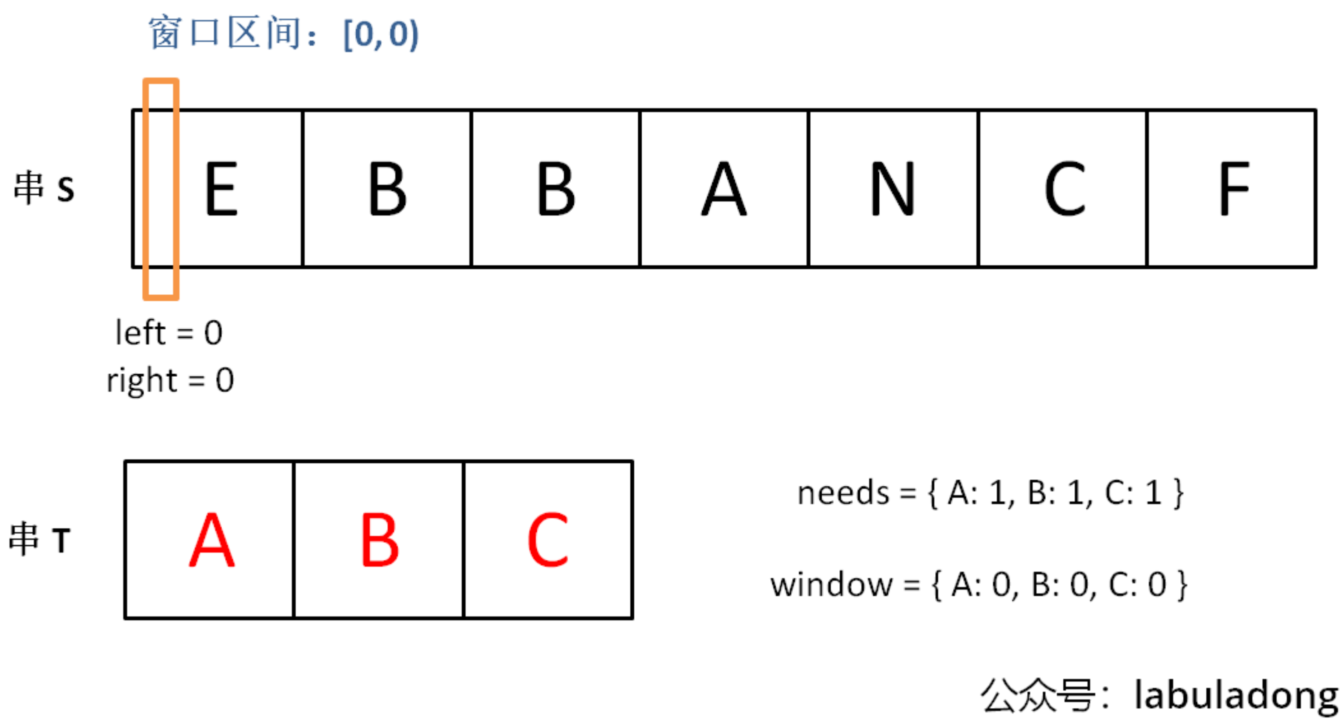
3、此时，我们停止增加 `right`，转而不断增加 `left` 指针缩小窗口 `[left, right)`，直到窗口中的字符串不再符合要求（不包含 T 中的所有字符了）。同时，每次增加 `left`，我们都要更新一轮结果。

4、重复第 2 和第 3 步，直到 `right` 到达字符串 S 的尽头。

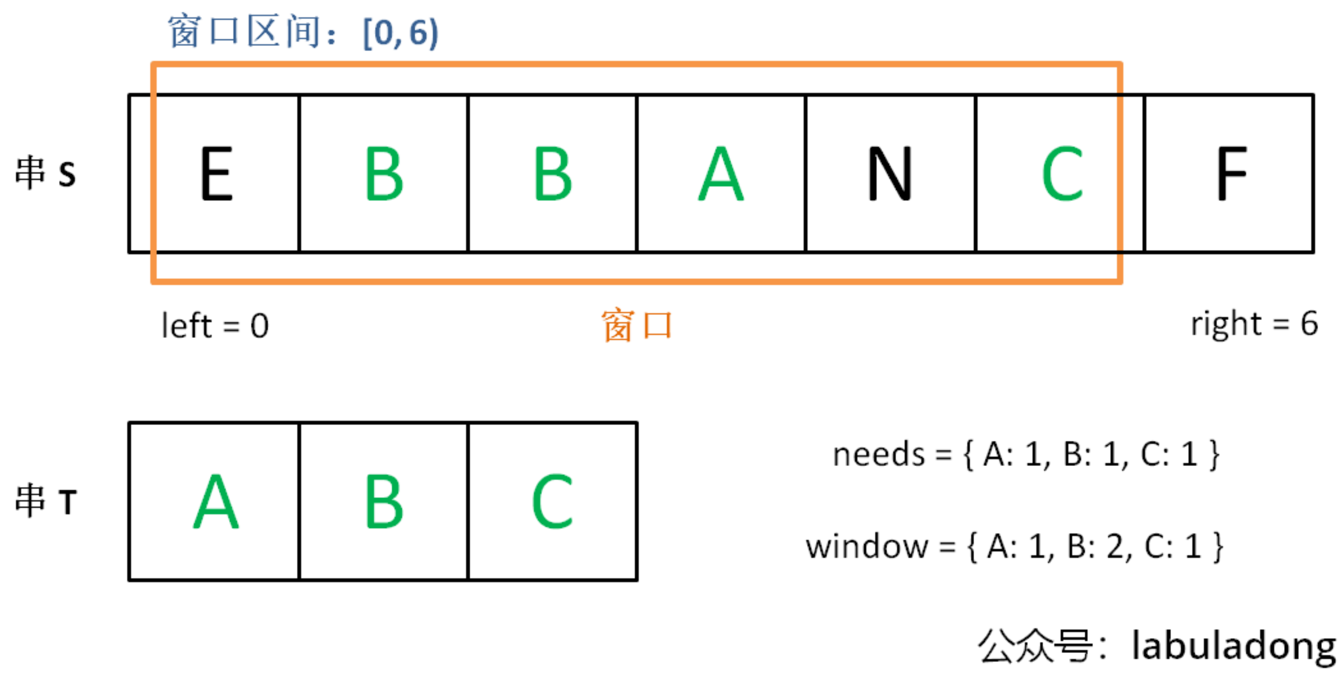
这个思路其实也不难，**第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解**，也就是最短的覆盖子串。左右指针轮流前进，窗口大小增增减减，窗口不断向右滑动，这就是「滑动窗口」这个名字的来历。

下面画图理解一下，`needs` 和 `window` 相当于计数器，分别记录 `T` 中字符出现次数和「窗口」中的相应字符的出现次数。

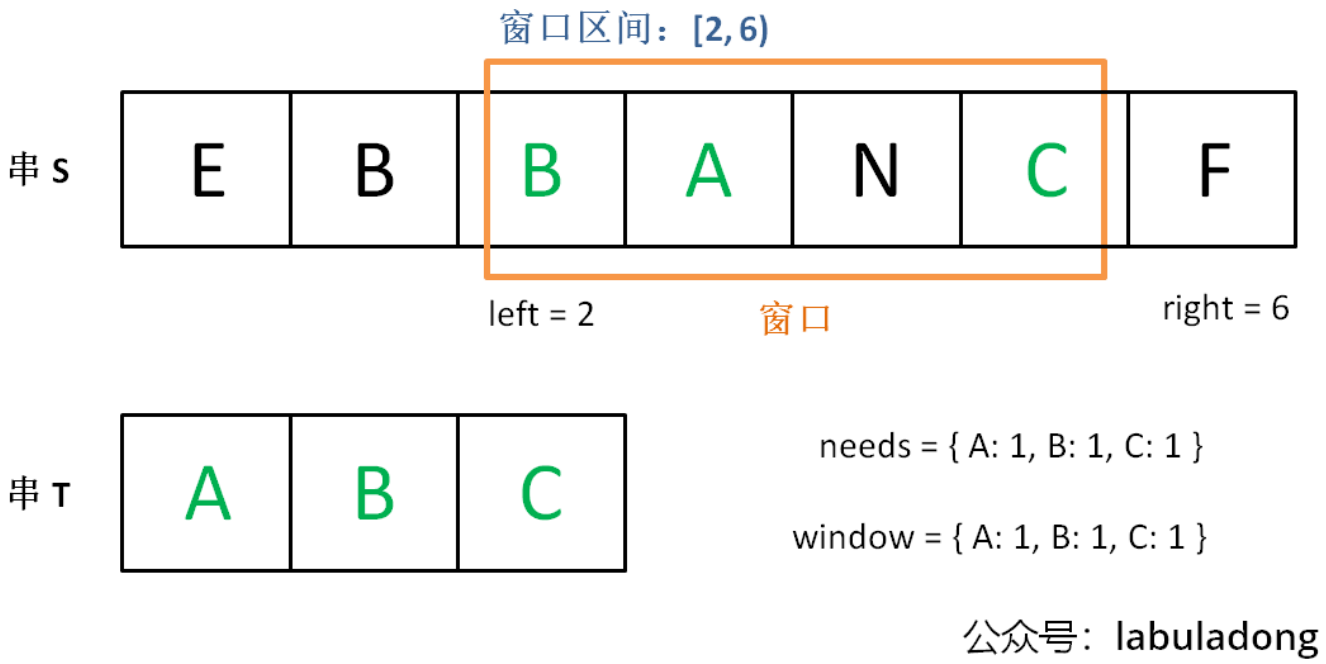
初始状态：



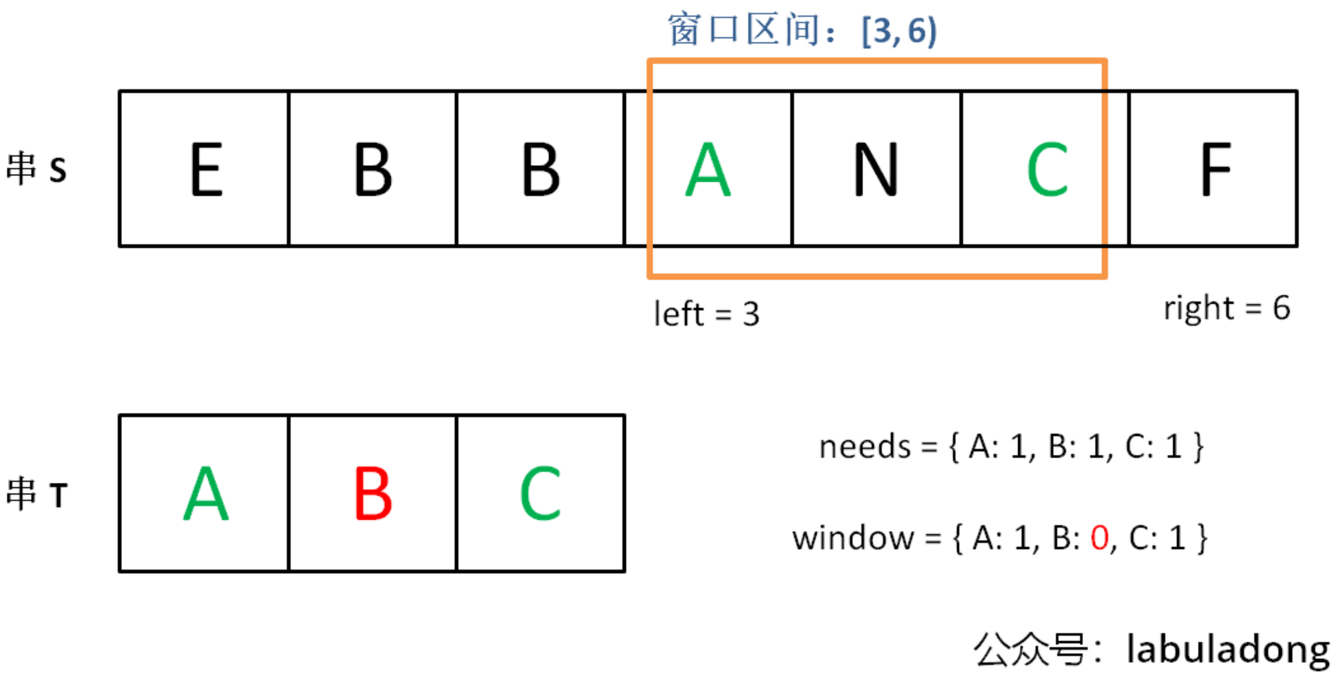
增加 `right`，直到窗口 `[left, right]` 包含了 `T` 中所有字符：



现在开始增加 `left`，缩小窗口 `[left, right]`：



直到窗口中的字符串不再符合要求，`left` 不再继续移动：



之后重复上述过程，先移动 `right`，再移动 `left`…… 直到 `right` 指针到达字符串 `S` 的末端，算法结束。

如果你能够理解上述过程，恭喜，你已经完全掌握了滑动窗口算法思想。**现在我们来看看这个滑动窗口代码框架怎么用**：

首先，初始化 `window` 和 `need` 两个哈希表，记录窗口中的字符和需要凑齐的字符：

```
unordered_map<char, int> need, window;
for (char c : t) need[c]++;
```

然后，使用 `left` 和 `right` 变量初始化窗口的两端，不要忘了，区间 `[left, right)` 是左闭右开的，所以初始情况下窗口没有包含任何元素：

```
int left = 0, right = 0;
int valid = 0;
while (right < s.size()) {
    // 开始滑动
}
```

其中 `valid` 变量表示窗口中满足 `need` 条件的字符个数，如果 `valid` 和 `need.size` 的大小相同，则说明窗口已满足条件，已经完全覆盖了串 `T`。

现在开始套模板，只需要思考以下四个问题：

1、当移动 `right` 扩大窗口，即加入字符时，应该更新哪些数据？

2、什么条件下，窗口应该暂停扩大，开始移动 `left` 缩小窗口？

3、当移动 `left` 缩小窗口，即移出字符时，应该更新哪些数据？

4、我们要的结果应该在扩大窗口时还是缩小窗口时进行更新？

如果一个字符进入窗口，应该增加 `window` 计数器；如果一个字符将移出窗口的时候，应该减少 `window` 计数器；当 `valid` 满足 `need` 时应该收缩窗口；应该在收缩窗口的时候更新最终结果。

下面是完整代码：

```
string minWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    // 记录最小覆盖子串的起始索引及长度
    int start = 0, len = INT_MAX;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c])
                valid++;
        }

        // 判断左侧窗口是否要收缩
        while (valid == need.size()) {
            // 在这里更新最小覆盖子串
            if (right - left < len) {
                start = left;
                len = right - left;
            }
```

```
                // d 是将移出窗口的字符
                char d = s[left];
                // 左移窗口
                left++;
                // 进行窗口内数据的一系列更新
                if (need.count(d)) {
                    if (window[d] == need[d])
                        valid--;
                    window[d]--;
                }
            }
        }
        // 返回最小覆盖子串
        return len == INT_MAX ?
            "" : s.substr(start, len);
    }
```

> PS：使用 Java 的读者要尤其警惕语言特性的陷阱。Java 的 Integer，String 等类型判定相等应该用 `equals` 方法而不能直接用等号 `==`，这是 Java包装类的一个隐晦细节。所以在左移窗口更新数据的时候，不能直接改写为 `window.get(d) == need.get(d)`，而要用 `window.get(d).equals(need.get(d))`，之后的题目代码同理。

需要注意的是，当我们发现某个字符在 `window` 的数量满足了 `need` 的需要，就要更新 `valid`，表示有一个字符已经满足要求。而且，你能发现，两次对窗口内数据的更新操作是完全对称的。

当 `valid == need.size()` 时，说明 `T` 中所有字符已经被覆盖，已经得到一个可行的覆盖子串，现在应该开始收缩窗口了，以便得到「最小覆盖子串」。

移动 `left` 收缩窗口时，窗口内的字符都是可行解，所以应该在收缩窗口的阶段进行最小覆盖子串的更新，以便从可行解中找到长度最短的最终结果。

至此，应该可以完全理解这套框架了，滑动窗口算法又不难，就是细节问题让人烦得很。**以后遇到滑动窗口算法，你就按照这框架写代码，保准没有 bug，还省事儿。**

下面就直接利用这套框架秒杀几道题吧，你基本上一眼就能看出思路了。

二、字符串排列

这是力扣第 567 题「字符串的排列」，难度中等：

567. 字符串的排列 labuladong 题解 思路

难度 中等 575

给你两个字符串 `s1` 和 `s2` ，写一个函数来判断 `s2` 是否包含 `s1` 的排列。如果是，返回 `true` ；否则，返回 `false` 。

换句话说， `s1` 的排列之一是 `s2` 的 **子串** 。

示例 1：

```
输入：s1 = "ab" s2 = "eidbaooo"
输出：true
解释：s2 包含 s1 的排列之一 ("ba").
```

示例 2：

```
输入：s1= "ab" s2 = "eidboaoo"
输出：false
```

注意哦，输入的 `s1` 是可以包含重复字符的，所以这个题难度不小。

这种题目，是明显的滑动窗口算法，**相当给你一个 `S` 和一个 `T`，请问你 `S` 中是否存在一个子串，包含 `T` 中所有字符且不包含其他字符**？

首先，先复制粘贴之前的算法框架代码，然后明确刚才提出的 4 个问题，即可写出这道题的答案：

```
// 判断 s 中是否存在 t 的排列
bool checkInclusion(string t, string s) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    while (right < s.size()) {
        char c = s[right];
        right++;
        // 进行窗口内数据的一系列更新
        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c])
                valid++;
```

```
        }

        // 判断左侧窗口是否要收缩
        while (right - left >= t.size()) {
            // 在这里判断是否找到了合法的子串
            if (valid == need.size())
                return true;
            char d = s[left];
            left++;
            // 进行窗口内数据的一系列更新
            if (need.count(d)) {
                if (window[d] == need[d])
                    valid--;
                window[d]--;
            }
        }
    }
    // 未找到符合条件的子串
    return false;
}
```

对于这道题的解法代码，基本上和最小覆盖子串一模一样，只需要改变两个地方：

1、本题移动 `left` 缩小窗口的时机是窗口大小大于 `t.size()` 时，应为排列嘛，显然长度应该是一样的。

2、当发现 `valid == need.size()` 时，就说明窗口中就是一个合法的排列，所以立即返回 `true`。

至于如何处理窗口的扩大和缩小，和最小覆盖子串完全相同。

三、找所有字母异位词

这是力扣第 438 题「找到字符串中所有字母异位词」，难度中等：

438. 找到字符串中所有字母异位词 labuladong 题解 思路

难度 中等 728 收藏 分享 切换为英文 接收动态 反馈

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 **异位词** 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。

示例 1:

```
输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

呵呵，这个所谓的字母异位词，不就是排列吗，搞个高端的说法就能糊弄人了吗？**相当于，输入一个串 S，一个串 T，找到 S 中所有 T 的排列，返回它们的起始索引。**

直接默写一下框架，明确刚才讲的 4 个问题，即可秒杀这道题：

```
vector<int> findAnagrams(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    vector<int> res; // 记录结果
    while (right < s.size()) {
        char c = s[right];
        right++;
        // 进行窗口内数据的一系列更新
        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c])
                valid++;
        }
        // 判断左侧窗口是否要收缩
        while (right - left >= t.size()) {
            // 当窗口符合条件时，把起始索引加入 res
            if (valid == need.size())
                res.push_back(left);
            char d = s[left];
            left++;
            // 进行窗口内数据的一系列更新
            if (need.count(d)) {
                if (window[d] == need[d])
                    valid--;
                window[d]--;
            }
        }
    }
    return res;
}
```

跟寻找字符串的排列一样，只是找到一个合法异位词（排列）之后将起始索引加入 res 即可。

四、最长无重复子串

这是力扣第 3 题「无重复字符的最长子串」，难度中等：

3. 无重复字符的最长子串 labuladong 题解 思路

难度 中等 6947

给定一个字符串 `s` ，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

```
输入: s = "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

示例 2:

```
输入: s = "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
```

这个题终于有了点新意，不是一套框架就出答案，不过反而更简单了，稍微改一改框架就行了：

```
int lengthOfLongestSubstring(string s) {
    unordered_map<char, int> window;

    int left = 0, right = 0;
    int res = 0; // 记录结果
    while (right < s.size()) {
        char c = s[right];
        right++;
        // 进行窗口内数据的一系列更新
        window[c]++;
        // 判断左侧窗口是否要收缩
        while (window[c] > 1) {
            char d = s[left];
            left++;
            // 进行窗口内数据的一系列更新
            window[d]--;
        }
        // 在这里更新答案
        res = max(res, right - left);
    }
```

```
        return res;
    }
```

这就是变简单了，连 need 和 valid 都不需要，而且更新窗口内数据也只需要简单的更新计数器 window 即可。

当 window[c] 值大于 1 时，说明窗口中存在重复字符，不符合条件，就该移动 left 缩小窗口了嘛。

唯一需要注意的是，在哪里更新结果 res 呢？我们要的是最长无重复子串，哪一个阶段可以保证窗口中的字符串是没有重复的呢？

这里和之前不一样，要在收缩窗口完成后更新 res，因为窗口收缩的 while 条件是存在重复元素，换句话说收缩完成后一定保证窗口中没有重复嘛。

五、最后总结

建议背诵并默写这套框架，顺便背诵一下文章开头的那首诗。以后就再也不怕子串、子数组问题了好吧。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



我写了首诗，把二分搜索变成了默写题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

704. 二分查找（简单）

34. 在排序数组中查找元素的第一个和最后一个位置（中等）

本文是前文 二分搜索详解 的修订版，添加了对二分搜索算法更详细的分析。

先给大家讲个笑话乐呵一下：

有一天阿东到图书馆借了 N 本书，出图书馆的时候，警报响了，于是保安把阿东拦下，要检查一下哪本书没有登记出借。阿东正准备把每一本书在报警器下过一下，以找出引发警报的书，但是保安露出不屑的眼神：你连二分查找都不会吗？于是保安把书分成两堆，让第一堆过一下报警器，报警器响；于是再把这堆书分成两堆…… 最终，检测了 logN 次之后，保安成功的找到了那本引起警报的书，露出了得意和嘲讽的笑容。于是阿东背着剩下的书走了。

从此，图书馆丢了 N - 1 本书。

二分查找并不简单，Knuth 大佬（发明 KMP 算法的那位）都说二分查找：**思路很简单，细节是魔鬼**。很多人喜欢拿整型溢出的 bug 说事儿，但是二分查找真正的坑根本就不是那个细节问题，而是在于到底要给 `mid` 加一还是减一，while 里到底用 `<=` 还是 `<`。

你要是没有正确理解这些细节，写二分肯定就是玄学编程，有没有 bug 只能靠菩萨保佑。**我特意写了一首诗来歌颂该算法，概括本文的主要内容，建议保存**：

二分搜索升天词
作者：labuladong

二分搜索不好记，左右边界让人迷。
小于等于变小于，mid 加一又减一。
就算这样还没完，return 应否再 -1？
信心满满刷力扣，AC 比率二十一。
我本将心向明月，奈何明月照沟渠！
问君能有几多愁？恰似深情喂了狗。

labuladong从天降，一同手撕算法题。
赠君一法写二分，不用拜佛与念经。
管他左侧还右侧，搜索区间定乾坤。

搜索一个元素时，搜索区间两端闭。
while 条件带等号，否则需要打补丁。
if 相等就返回，其他的事甭操心。
mid 必须加减一，因为区间两端闭。
while结束就凉了，凄凄惨惨返 -1。

搜索左右边界时，搜索区间要阐明。
左闭右开最常见，其余逻辑便自明：
while要用小于号，这样才能不漏掉。
if 相等别返回，利用 mid 锁边界。
mid 加一或减一？要看区间开或闭。
while结束不算完，因为你还没返回。
索引可能出边界，if 检查保平安。

左闭右开最常见，难道常见就合理？
labuladong不信邪，偏要改成两端闭。
搜索区间记于心，或开或闭有何异？
二分搜索三变体，逻辑统一容易记。
一套框架改两行，胜过千言和万语。

此等神人何处寻？全靠缘分不可期！
labuladong公众号，开启算法新天地。
关注标星加分享，“下次一定”不可取。

本文就来探究几个最常用的二分查找场景：寻找一个数、寻找左侧边界、寻找右侧边界。而且，我们就是要深入细节，比如不等号是否应该带等号，`mid` 是否应该加一等等。分析这些细节的差异以及出现这些差异的原因，保证你能灵活准确地写出正确的二分查找算法。

零、二分查找框架

```java
int binarySearch(int[] nums, int target) {
    int left = 0, right = ...;
```

```
    while(...) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            ...
        } else if (nums[mid] < target) {
            left = ...
        } else if (nums[mid] > target) {
            right = ...
        }
    }
    return ...;
}
```

分析二分查找的一个技巧是：不要出现 else，而是把所有情况用 else if 写清楚，这样可以清楚地展现所有细节。本文都会使用 else if，旨在讲清楚，读者理解后可自行简化。

其中 `...` 标记的部分，就是可能出现细节问题的地方，当你见到一个二分查找的代码时，首先注意这几个地方。后文用实例分析这些地方能有什么样的变化。

另外声明一下，计算 mid 时需要防止溢出，代码中 `left + (right - left) / 2` 就和 `(left + right) / 2` 的结果相同，但是有效防止了 `left` 和 `right` 太大直接相加导致溢出。

一、寻找一个数（基本的二分搜索）

这个场景是最简单的，可能也是大家最熟悉的，即搜索一个数，如果存在，返回其索引，否则返回 -1。

```
int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1; // 注意

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1; // 注意
        else if (nums[mid] > target)
            right = mid - 1; // 注意
    }
    return -1;
}
```

这段代码可以解决力扣第 704 题「二分查找」，但我们深入探讨一下其中的细节。

1、为什么 while 循环的条件中是 <=，而不是 <?

答：因为初始化 `right` 的赋值是 `nums.length - 1`，即最后一个元素的索引，而不是 `nums.length`。

这二者可能出现在不同功能的二分查找中，区别是：前者相当于两端都闭区间 `[left, right]`，后者相当于左闭右开区间 `[left, right)`，因为索引大小为 `nums.length` 是越界的。

我们这个算法中使用的是前者 [left, right] 两端都闭的区间。**这个区间其实就是每次进行搜索的区间。**

什么时候应该停止搜索呢？当然，找到了目标值的时候可以终止：

```
if(nums[mid] == target)
    return mid;
```

但如果没找到，就需要 while 循环终止，然后返回 -1。那 while 循环什么时候应该终止？**搜索区间为空的时候应该终止**，意味着你没得找了，就等于没找到嘛。

while(left <= right) 的终止条件是 left == right + 1，写成区间的形式就是 [right + 1, right]，或者带个具体的数字进去 [3, 2]，可见**这时候区间为空**，因为没有数字既大于等于 3 又小于等于 2 的吧。所以这时候 while 循环终止是正确的，直接返回 -1 即可。

while(left < right) 的终止条件是 left == right，写成区间的形式就是 [right, right]，或者带个具体的数字进去 [2, 2]，**这时候区间非空**，还有一个数 2，但此时 while 循环终止了。也就是说这区间 [2, 2] 被漏掉了，索引 2 没有被搜索，如果这时候直接返回 -1 就是错误的。

当然，如果你非要用 while(left < right) 也可以，我们已经知道了出错的原因，就打个补丁好了：

```
//...
while(left < right) {
    // ...
}
return nums[left] == target ? left : -1;
```

2、为什么 left = mid + 1，right = mid - 1？我看有的代码是 right = mid 或者 left = mid，没有这些加加减减，到底怎么回事，怎么判断？

答：这也是二分查找的一个难点，不过只要你能理解前面的内容，就能够很容易判断。

刚才明确了「搜索区间」这个概念，而且本算法的搜索区间是两端都闭的，即 [left, right]。那么当我们发现索引 mid 不是要找的 target 时，下一步应该去搜索哪里呢？

当然是去搜索 [left, mid-1] 或者 [mid+1, right] 对不对？**因为 mid 已经搜索过，应该从搜索区间中去除**。

3、此算法有什么缺陷？

答：至此，你应该已经掌握了该算法的所有细节，以及这样处理的原因。但是，这个算法存在局限性。

比如说给你有序数组 nums = [1,2,2,2,3]，target 为 2，此算法返回的索引是 2，没错。但是如果我想得到 target 的左侧边界，即索引 1，或者我想得到 target 的右侧边界，即索引 3，这样的话此算法是无法处理的。

这样的需求很常见，**你也许会说，找到一个 target，然后向左或向右线性搜索不行吗？可以，但是不好，因为这样难以保证二分查找对数级的复杂度了**。

我们后续的算法就来讨论这两种二分查找的算法。

二、寻找左侧边界的二分搜索

以下是最常见的代码形式，其中的标记是需要注意的细节：

```
int left_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0;
    int right = nums.length; // 注意

    while (left < right) { // 注意
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            right = mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid; // 注意
        }
    }
    return left;
}
```

1、为什么 while 中是 `<` 而不是 `<=`?

答：用相同的方法分析，因为 `right = nums.length` 而不是 `nums.length - 1`。因此每次循环的「搜索区间」是 `[left, right)` 左闭右开。

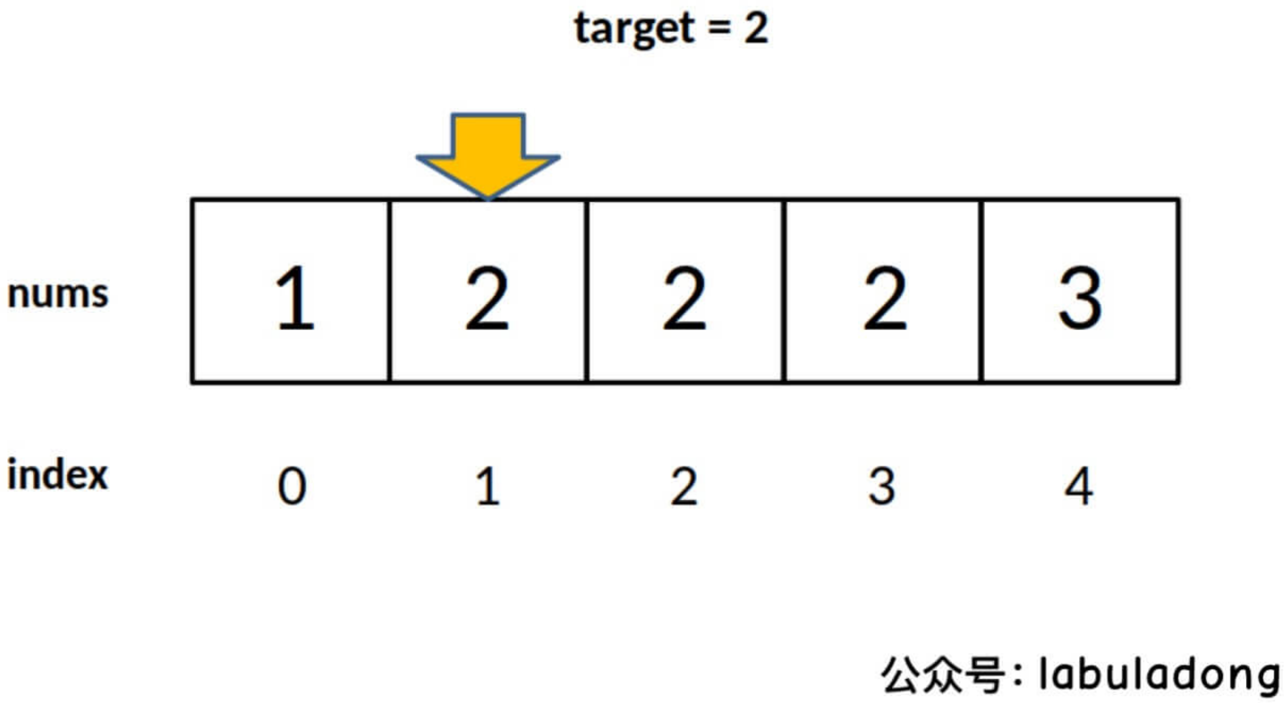
`while(left < right)` 终止的条件是 `left == right`，此时搜索区间 `[left, left)` 为空，所以可以正确终止。

> PS：这里先要说一个搜索左右边界和上面这个算法的一个区别，也是很多读者问的：**刚才的 `right` 不是 `nums.length - 1` 吗，为啥这里非要写成 `nums.length` 使得「搜索区间」变成左闭右开呢？**

因为对于搜索左右侧边界的二分查找，这种写法比较普遍，我就拿这种写法举例了，保证你以后遇到这类代码可以理解。你非要用两端都闭的写法反而更简单，我会在后面写相关的代码，把三种二分搜索都用一种两端都闭的写法统一起来，你耐心往后看就行了。

2、为什么没有返回 -1 的操作？如果 `nums` 中不存在 `target` 这个值，怎么办？

答：因为要一步一步来，先理解一下这个「左侧边界」有什么特殊含义：



对于这个数组，算法会返回索引 1。

这个索引 1 的含义可以解读为「nums 中小于 2 的元素有 1 个」。

比如对于有序数组 nums = [2,3,5,7], target = 1，算法会返回 0，含义是：nums 中小于 1 的元素有 0 个。

再比如说 nums = [2,3,5,7], target = 8，算法会返回 4，含义是：nums 中小于 8 的元素有 4 个。

> PS：对于 target 不存在 nums 中的情况，函数的返回值还可以有多种理解方式，详见 随机权重算法 中对二分搜索的运用。

综上可以看出，函数的返回值（即 left 变量的值）取值区间是闭区间 [0, nums.length]，所以我们简单添加两行代码就能在正确的时候 return -1：

```
while (left < right) {
    //...
}
// target 比所有数都大
if (left == nums.length) return -1;
// 类似之前算法的处理方式
return nums[left] == target ? left : -1;
```

3、为什么 left = mid + 1，right = mid？和之前的算法不一样？

答：这个很好解释，因为我们的「搜索区间」是 [left, right) 左闭右开，所以当 nums[mid] 被检测之后，下一步的搜索区间应该去掉 mid 分割成两个区间，即 [left, mid) 或 [mid + 1, right)。

4、为什么该算法能够搜索左侧边界？

答：关键在于对于 `nums[mid] == target` 这种情况的处理：

```
    if (nums[mid] == target)
        right = mid;
```

可见，找到 target 时不要立即返回，而是缩小「搜索区间」的上界 `right`，在区间 `[left, mid)` 中继续搜索，即不断向左收缩，达到锁定左侧边界的目的。

5、为什么返回 `left` 而不是 `right`？

答：都是一样的，因为 while 终止的条件是 `left == right`。

6、能不能想办法把 `right` 变成 `nums.length - 1`，也就是继续使用两边都闭的「搜索区间」？这样就可以和第一种二分搜索在某种程度上统一起来了。

答：当然可以，只要你明白了「搜索区间」这个概念，就能有效避免漏掉元素，随便你怎么改都行。下面我们严格根据逻辑来修改：

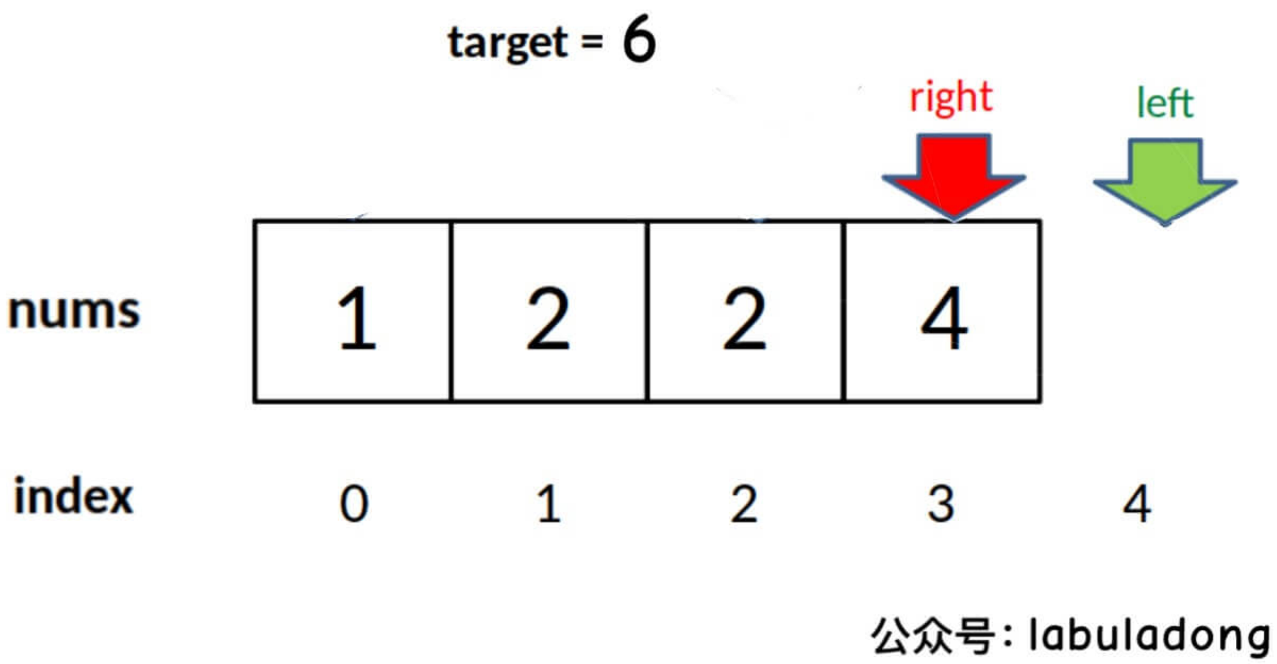
因为你非要让搜索区间两端都闭，所以 `right` 应该初始化为 `nums.length - 1`，while 的终止条件应该是 `left == right + 1`，也就是其中应该用 `<=`：

```
int left_bound(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        // if else ...
    }
```

因为搜索区间是两端都闭的，且现在是搜索左侧边界，所以 `left` 和 `right` 的更新逻辑如下：

```
if (nums[mid] < target) {
    // 搜索区间变为 [mid+1, right]
    left = mid + 1;
} else if (nums[mid] > target) {
    // 搜索区间变为 [left, mid-1]
    right = mid - 1;
} else if (nums[mid] == target) {
    // 收缩右侧边界
    right = mid - 1;
}
```

由于 while 的退出条件是 `left == right + 1`，所以当 `target` 比 `nums` 中所有元素都大时，会存在以下情况使得索引越界：



因此，最后返回结果的代码应该检查越界情况：

```
if (left >= nums.length || nums[left] != target)
    return -1;
return left;
```

至此，整个算法就写完了，完整代码如下：

```
int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    // 搜索区间为 [left, right]
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩右侧边界
            right = mid - 1;
        }
    }
    // 检查出界情况
    if (left >= nums.length || nums[left] != target) {
        return -1;
    }
```

```
    return left;
}
```

这样就和第一种二分搜索算法统一了，都是两端都闭的「搜索区间」，而且最后返回的也是 `left` 变量的值。只要把住二分搜索的逻辑，两种形式大家看自己喜欢哪种记哪种吧。

三、寻找右侧边界的二分查找

类似寻找左侧边界的算法，这里也会提供两种写法，还是先写常见的左闭右开的写法，只有两处和搜索左侧边界不同，已标注：

```
int right_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            left = mid + 1; // 注意
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid;
        }
    }
    return left - 1; // 注意
}
```

1、为什么这个算法能够找到右侧边界？

答：类似地，关键点还是这里：

```
if (nums[mid] == target) {
    left = mid + 1;
```

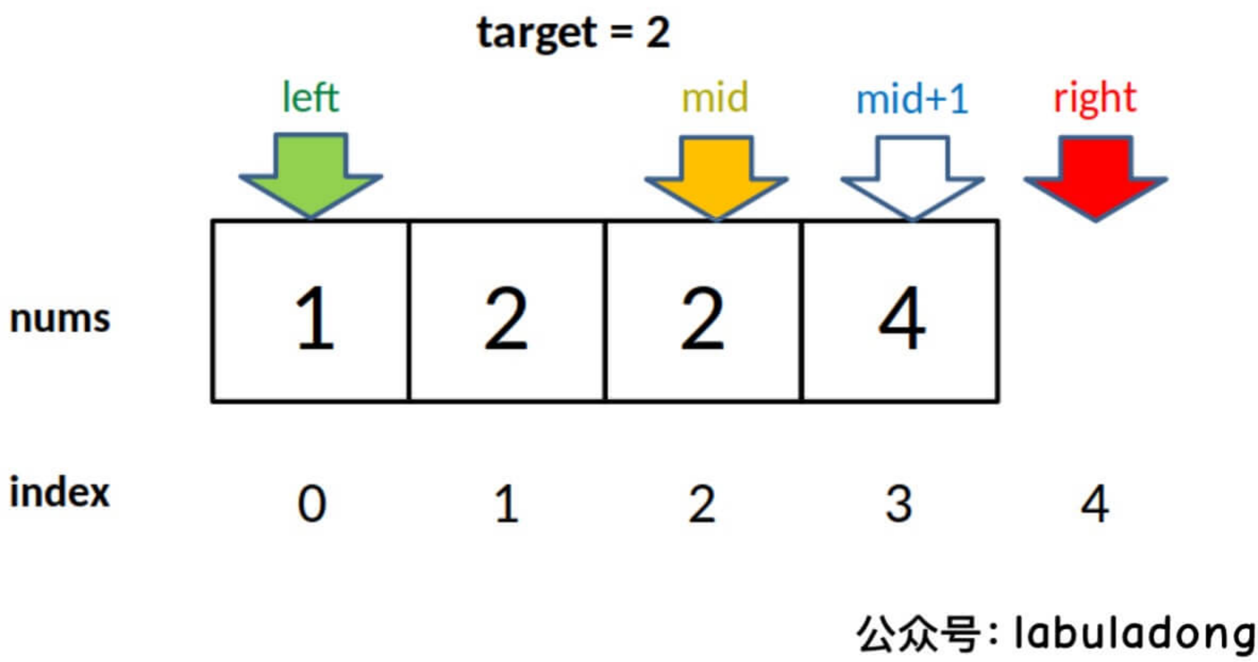
当 `nums[mid] == target` 时，不要立即返回，而是增大「搜索区间」的下界 `left`，使得区间不断向右收缩，达到锁定右侧边界的目的。

2、为什么最后返回 `left - 1` 而不像左侧边界的函数，返回 `left`？而且我觉得这里既然是搜索右侧边界，应该返回 `right` 才对。

答：首先，while 循环的终止条件是 `left == right`，所以 `left` 和 `right` 是一样的，你非要体现右侧的特点，返回 `right - 1` 好了。

至于为什么要减一，这是搜索右侧边界的一个特殊点，关键在这个条件判断：

```
if (nums[mid] == target) {
    left = mid + 1;
    // 这样想: mid = left - 1
```



因为我们对 `left` 的更新必须是 `left = mid + 1`，就是说 while 循环结束时，`nums[left]` 一定不等于 `target` 了，而 `nums[left-1]` 可能是 `target`。

至于为什么 `left` 的更新必须是 `left = mid + 1`，同左侧边界搜索，就不再赘述。

3、为什么没有返回 -1 的操作？如果 `nums` 中不存在 `target` 这个值，怎么办？

答：类似之前的左侧边界搜索，因为 while 的终止条件是 `left == right`，就是说 `left` 的取值范围是 `[0, nums.length]`，所以可以添加两行代码，正确地返回 -1：

```
while (left < right) {
    // ...
}
if (left == 0) return -1;
return nums[left-1] == target ? (left-1) : -1;
```

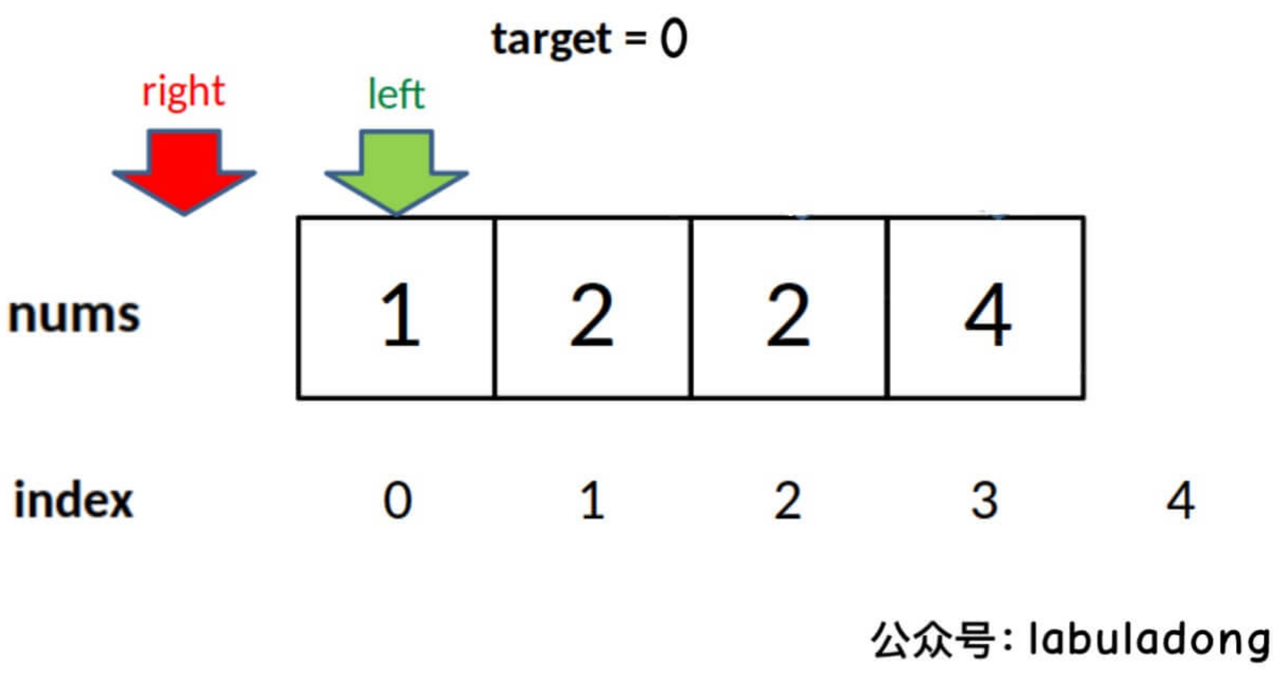
4、是否也可以把这个算法的「搜索区间」也统一成两端都闭的形式呢？这样这三个写法就完全统一了，以后就可以闭着眼睛写出来了。

答：当然可以，类似搜索左侧边界的统一写法，其实只要改两个地方就行了：

```
int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
```

```
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] == target) {
                // 这里改成收缩左侧边界即可
                left = mid + 1;
            }
        }
        // 这里改为检查 right 越界的情况，见下图
        if (right < 0 || nums[right] != target) {
            return -1;
        }
        return right;
    }
```

当 `target` 比所有元素都小时，`right` 会被减到 -1，所以需要在最后防止越界：



至此，搜索右侧边界的二分查找的两种写法也完成了，其实将「搜索区间」统一成两端都闭反而更容易记忆，你说是吧？

四、逻辑统一

有了搜索左右边界的二分搜索，你可以去解决力扣第 34 题「[在排序数组中查找元素的第一个和最后一个位置](#)」，

接下来梳理一下这些细节差异的因果逻辑：

第一个，最基本的二分查找算法：

```
因为我们初始化 right = nums.length - 1
所以决定了我们的「搜索区间」是 [left, right]
所以决定了 while (left <= right)
同时也决定了 left = mid+1 和 right = mid-1

因为我们只需找到一个 target 的索引即可
所以当 nums[mid] == target 时可以立即返回
```

第二个，寻找左侧边界的二分查找：

```
因为我们初始化 right = nums.length
所以决定了我们的「搜索区间」是 [left, right)
所以决定了 while (left < right)
同时也决定了 left = mid + 1 和 right = mid

因为我们需找到 target 的最左侧索引
所以当 nums[mid] == target 时不要立即返回
而要收紧右侧边界以锁定左侧边界
```

第三个，寻找右侧边界的二分查找：

```
因为我们初始化 right = nums.length
所以决定了我们的「搜索区间」是 [left, right)
所以决定了 while (left < right)
同时也决定了 left = mid + 1 和 right = mid

因为我们需找到 target 的最右侧索引
所以当 nums[mid] == target 时不要立即返回
而要收紧左侧边界以锁定右侧边界

又因为收紧左侧边界时必须 left = mid + 1
所以最后无论返回 left 还是 right，必须减一
```

对于寻找左右边界的二分搜索，常见的手法是使用左闭右开的「搜索区间」，**我们还根据逻辑将「搜索区间」全都统一成了两端都闭，便于记忆，只要修改两处即可变化出三种写法**：

```
int binary_search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while(left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if(nums[mid] == target) {
            // 直接返回
```

```
                return mid;
            }
        }
        // 直接返回
        return -1;
    }

    int left_bound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] == target) {
                // 别返回，锁定左侧边界
                right = mid - 1;
            }
        }
        // 最后要检查 left 越界的情况
        if (left >= nums.length || nums[left] != target) {
            return -1;
        }
        return left;
    }

    int right_bound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] == target) {
                // 别返回，锁定右侧边界
                left = mid + 1;
            }
        }
        // 最后要检查 right 越界的情况
        if (right < 0 || nums[right] != target) {
            return -1;
        }
        return right;
    }
```

如果以上内容你都能理解，那么恭喜你，二分查找算法的细节不过如此。

通过本文，你学会了：

1、分析二分查找代码时，不要出现 else，全部展开成 else if 方便理解。

2、注意「搜索区间」和 while 的终止条件，如果存在漏掉的元素，记得在最后检查。

3、如需定义左闭右开的「搜索区间」搜索左右边界，只要在 `nums[mid] == target` 时做修改即可，搜索右侧时需要减一。

4、如果将「搜索区间」全都统一成两端都闭，好记，只要稍改 `nums[mid] == target` 条件处的代码和返回的逻辑即可，**推荐拿小本本记下，作为二分搜索模板**。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

二分搜索题型套路分析

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

875. 爱吃香蕉的珂珂（中等）

1011. 在D天内送达包裹的能力（中等）

我们前文 我写了首诗，把二分搜索变成了默写题 详细介绍了二分搜索的细节问题，探讨了「搜索一个元素」，「搜索左侧边界」，「搜索右侧边界」这三个情况，教你如何写出正确无 bug 的二分搜索算法。

但是前文总结的二分搜索代码框架仅仅局限于「在有序数组中搜索指定元素」这个基本场景，具体的算法问题没有这么直接，可能你都很难看出这个问题能够用到二分搜索。

对于二分搜索算法在具体问题中的运用，前文 二分搜索的运用（一） 和前文 二分搜索的运用（二） 有过介绍，但是还没有抽象出来一个具体的套路框架。

所以本文就来总结一套二分搜索算法运用的框架套路，帮你在遇到二分搜索算法相关的实际问题时，能够有条理地思考分析，步步为营，写出答案。

警告：本文略长略硬核，建议清醒时学习。

原始的二分搜索代码

二分搜索的原型就是在「**有序数组**」中搜索一个元素 `target`，返回该元素对应的索引。

如果该元素不存在，那可以返回一个什么特殊值，这种细节问题只要微调算法实现就可实现。

还有一个重要的问题，如果「**有序数组**」中存在多个 `target` 元素，那么这些元素肯定挨在一起，这里就涉及到算法应该返回最左侧的那个 `target` 元素的索引还是最右侧的那个 `target` 元素的索引，也就是所谓的「搜索左侧边界」和「搜索右侧边界」，这个也可以通过微调算法的代码来实现。

我们前文 我写了首诗，把二分搜索变成了默写题 详细探讨了上述问题，对这块还不清楚的读者建议复习前文，已经搞清楚基本二分搜索算法的读者可以继续看下去。

在具体的算法问题中，常用到的是「搜索左侧边界」和「搜索右侧边界」这两种场景，很少有让你单独「搜索一个元素」。

因为算法题一般都让你求最值，比如前文 二分搜索的运用（一） 中说的例题让你求吃香蕉的「最小速度」，让你求轮船的「最低运载能力」，前文 二分搜索的运用（二） 讲的题就更魔幻了，让你使每个子数组之和的「最大值最小」。

求最值的过程，必然是搜索一个边界的过程，所以后面我们就详细分析一下这两种搜索边界的二分算法代码。

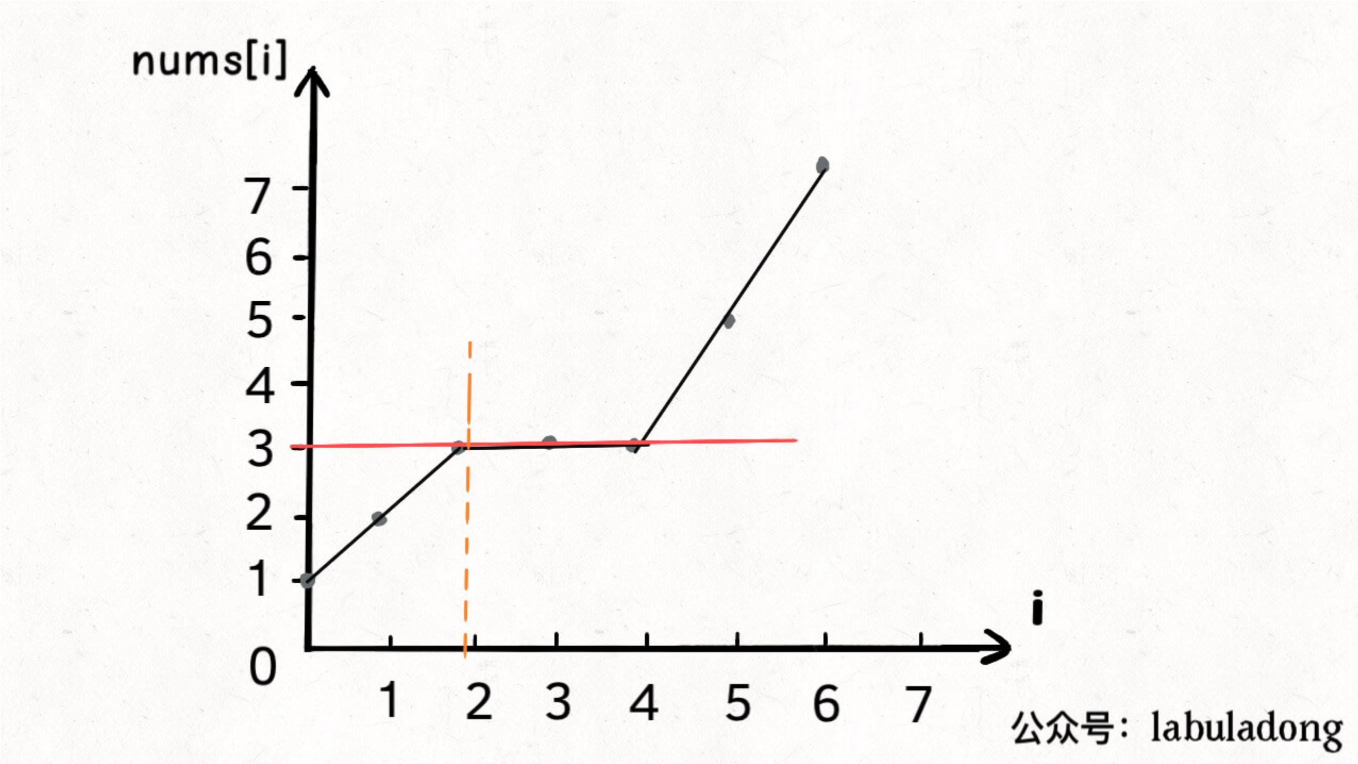
「搜索左侧边界」的二分搜索算法的具体代码实现如下：

```
// 搜索左侧边界
int left_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            // 当找到 target 时，收缩右侧边界
            right = mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid;
        }
    }
    return left;
}
```

假设输入的数组 `nums = [1,2,3,3,3,5,7]`，想搜索的元素 `target = 3`，那么算法就会返回索引 2。

如果画一个图，就是这样：

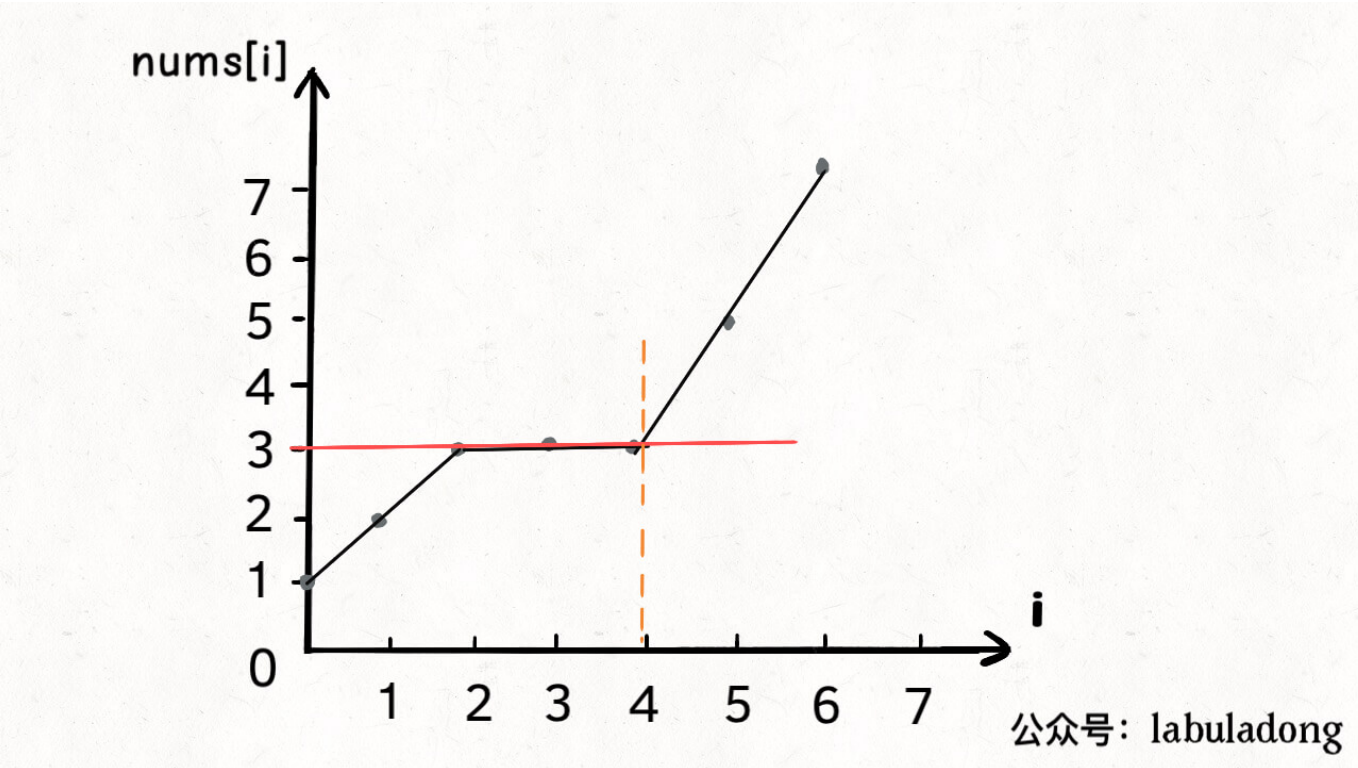


「搜索右侧边界」的二分搜索算法的具体代码实现如下：

```
// 搜索右侧边界
int right_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            // 当找到 target 时，收缩左侧边界
            left = mid + 1;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid;
        }
    }
    return left - 1;
}
```

输入同上，那么算法就会返回索引 4，如果画一个图，就是这样：



好，上述内容都属于复习，我想读到这里的读者应该都能理解。记住上述的图像，所有能够抽象出上述图像的问题，都可以使用二分搜索解决。

二分搜索问题的泛化

什么问题可以运用二分搜索算法技巧？

首先，你要从题目中抽象出一个自变量 `x`，一个关于 `x` 的函数 `f(x)`，以及一个目标值 `target`。

同时，`x, f(x), target` 还要满足以下条件：

1、`f(x)` 必须是在 `x` 上的单调函数（单调增单调减都可以）。

2、题目是让你计算满足约束条件 `f(x) == target` 时的 `x` 的值。

上述规则听起来有点抽象，来举个具体的例子：

给你一个升序排列的有序数组 `nums` 以及一个目标元素 `target`，请你计算 `target` 在数组中的索引位置，如果有多个目标元素，返回最小的索引。

这就是「搜索左侧边界」这个基本题型，解法代码之前都写了，但这里面 `x, f(x), target` 分别是什么呢？

我们可以把数组中元素的索引认为是自变量 `x`，函数关系 `f(x)` 就可以这样设定：

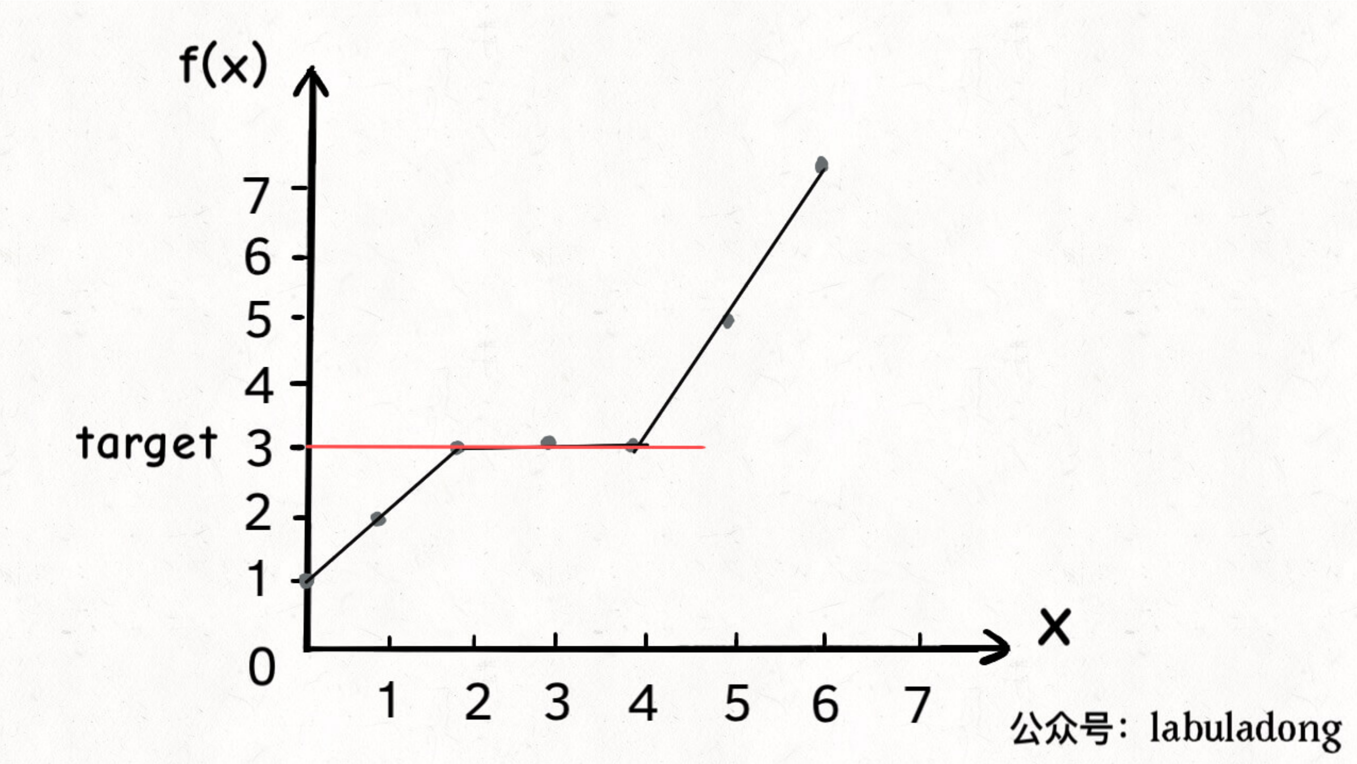
```java
// 函数 f(x) 是关于自变量 x 的单调递增函数
// 入参 nums 是不会改变的，所以可以忽略，不算自变量
int f(int x, int[] nums) {
    return nums[x];
}
```

其实这个函数 f 就是在访问数组 nums，因为题目给我们的数组 nums 是升序排列的，所以函数 f(x) 就是在 x 上单调递增的函数。

最后，题目让我们求什么来着？是不是让我们计算元素 target 的最左侧索引？

是不是就相当于在问我们「满足 f(x) == target 的 x 的最小值是多少」？

画个图，如下：



如果遇到一个算法问题，能够把它抽象成这幅图，就可以对它运用二分搜索算法。

算法代码如下：

```
// 函数 f 是关于自变量 x 的单调递增函数
int f(int x, int[] nums) {
    return nums[x];
}

int left_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (f(mid, nums) == target) {
            // 当找到 target 时，收缩右侧边界
            right = mid;
        } else if (f(mid, nums) < target) {
            left = mid + 1;
        } else if (f(mid, nums) > target) {
            right = mid;
        }
```

```
    }
    return left;
}
```

这段代码把之前的代码微调了一下，把直接访问 `nums[mid]` 套了一层函数 `f`，其实就是多此一举，但是，这样能抽象出二分搜索思想在具体算法问题中的框架。

运用二分搜索的套路框架

想要运用二分搜索解决具体的算法问题，可以从以下代码框架着手思考：

```
// 函数 f 是关于自变量 x 的单调函数
int f(int x) {
    // ...
}

// 主函数，在 f(x) == target 的约束下求 x 的最值
int solution(int[] nums, int target) {
    if (nums.length == 0) return -1;
    // 问自己：自变量 x 的最小值是多少？
    int left = ...;
    // 问自己：自变量 x 的最大值是多少？
    int right = ... + 1;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (f(mid) == target) {
            // 问自己：题目是求左边界还是右边界？
            // ...
        } else if (f(mid) < target) {
            // 问自己：怎么让 f(x) 大一点？
            // ...
        } else if (f(mid) > target) {
            // 问自己：怎么让 f(x) 小一点？
            // ...
        }
    }
    return left;
}
```

具体来说，想要用二分搜索算法解决问题，分为以下几步：

1、确定 `x, f(x), target` 分别是什么，并写出函数 `f` 的代码。

2、找到 `x` 的取值范围作为二分搜索的搜索区间，初始化 `left` 和 `right` 变量。

3、根据题目的要求，确定应该使用搜索左侧还是搜索右侧的二分搜索算法，写出解法代码。

下面用几道例题来讲解这个流程。

应合作方要求，本文不便在此发布，请扫码关注回复关键词「二分」查看：

田忌赛马背后的算法决策

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

870. 优势洗牌（中等）

田忌赛马的故事大家应该都听说过：

田忌和齐王赛马，两人的马分上中下三等，如果同等级的马对应着比赛，田忌赢不了齐王。但是田忌遇到了孙膑，孙膑就教他用自己的下等马对齐王的上等马，再用自己的上等马对齐王的中等马，最后用自己的中等马对齐王的下等马，结果三局两胜，田忌赢了。

当然，这段历史也挺有意思的，那个讽齐王纳谏，自恋的不行的邹忌和田忌是同一时期的人，他俩后来就杠上了。不过这是题外话，我们这里就打住。

以前学到田忌赛马课文的时，我就在想，如果不是三匹马比赛，而是一百匹马比赛，孙膑还能不能合理地安排比赛的顺序，赢得齐王呢？

当时没想出什么好的点子，只觉得这里面最核心问题是要尽可能让自己占便宜，让对方吃亏。总结来说就是，**打得过就打，打不过就拿自己的垃圾和对方的精锐互换**。

不过，我一直没具体把这个思路实现出来，直到最近刷到力扣第 870 题「优势洗牌」，一眼就发现这是田忌赛马问题的加强版：

给你输入两个**长度相等**的数组 `nums1` 和 `nums2`，请你重新组织 `nums1` 中元素的位置，使得 `nums1` 的「优势」最大化。

如果 `nums1[i] > nums2[i]`，就是说 `nums1` 在索引 `i` 上对 `nums2[i]` 有「优势」。优势最大化也就是说让你重新组织 `nums1`，**尽可能多的让 `nums[i] > nums2[i]`**。

算法签名如下：

```java
int[] advantageCount(int[] nums1, int[] nums2);
```

比如输入：

`nums1 = [12,24,8,32] nums2 = [13,25,32,11]`

你的算法应该返回 `[24,32,8,12]`，因为这样排列 `nums1` 的话有三个元素都有「优势」。

这就像田忌赛马的情景，`nums1` 就是田忌的马，`nums2` 就是齐王的马，数组中的元素就是马的战斗力，你就是孙膑，展示你真正的技术吧。

仔细想想，这个题的解法还是有点扑朔迷离的。什么时候应该放弃抵抗去送人头，什么时候应该硬刚？这里面应该有一种算法策略来最大化「优势」。

送人头一定是迫不得已而为之的权宜之计，否则隔壁田忌就要开语音骂你菜了。只有田忌的上等马比不过齐王的上等马时，才会用下等马去和齐王的上等马互换。

对于比较复杂的问题，可以尝试从特殊情况考虑。

你想，谁应该去应对齐王最快的马？肯定是田忌最快的那匹马，我们简称一号选手。

如果田忌的一号选手比不过齐王的一号选手，那其他马肯定是白给了，显然这种情况肯定应该用田忌垫底的马去送人头，降低己方损失，保存实力，增加接下来比赛的胜率。

但如果田忌的一号选手能比得过齐王的一号选手，那就和齐王硬刚好了，反正这把田忌可以赢。

你也许说，这种情况下说不定田忌的二号选手也能干得过齐王的一号选手？如果可以的话，让二号选手去对决齐王的一号选手，不是更节约？

就好比，如果考 60 分就能过的话，何必考 90 分？每多考一分就亏一分，刚刚好卡在 60 分是最划算的。

这种节约的策略是没问题的，但是没有必要。这也是本题有趣的地方，需要绕个脑筋急转弯：

我们暂且把田忌的一号选手称为 `T1`，二号选手称为 `T2`，齐王的一号选手称为 `Q1`。

如果 `T2` 能赢 `Q1`，你试图保存己方实力，让 `T2` 去战 `Q1`，把 `T1` 留着是为了对付谁？

显然，你担心齐王还有战力大于 `T2` 的马，可以让 `T1` 去对付。

但是你仔细想想，现在 `T2` 已经是可以战胜 `Q1` 的，`Q1` 可是齐王的最快的马耶，齐王剩下的那些马里，怎么可能还有比 `T2` 更强的马？

所以，没必要节约，最后我们得出的策略就是：

将齐王和田忌的马按照战斗力排序，然后按照排名一一对比。如果田忌的马能赢，那就比赛，如果赢不了，那就换个垫底的来送人头，保存实力。

上述思路的代码逻辑如下：

```
int n = nums1.length;

sort(nums1); // 田忌的马
sort(nums2); // 齐王的马

// 从最快的马开始比
for (int i = n - 1; i >= 0; i--) {
    if (nums1[i] > nums2[i]) {
        // 比得过，跟他比
    } else {
```

```
            // 比不过，换个垫底的来送人头
        }
    }
```

根据这个思路，我们需要对两个数组排序，但是 nums2 中元素的顺序不能改变，因为计算结果的顺序依赖 nums2 的顺序，所以不能直接对 nums2 进行排序，而是利用其他数据结构来辅助。

同时，最终的解法还用到前文 双指针技巧汇总 总结的双指针算法模板，用以处理「送人头」的情况：

```
int[] advantageCount(int[] nums1, int[] nums2) {
    int n = nums1.length;
    // 给 nums2 降序排序
    PriorityQueue<int[]> maxpq = new PriorityQueue<>(
        (int[] pair1, int[] pair2) -> {
            return pair2[1] - pair1[1];
        }
    );
    for (int i = 0; i < n; i++) {
        maxpq.offer(new int[]{i, nums2[i]});
    }
    // 给 nums1 升序排序
    Arrays.sort(nums1);

    // nums1[left] 是最小值，nums1[right] 是最大值
    int left = 0, right = n - 1;
    int[] res = new int[n];

    while (!maxpq.isEmpty()) {
        int[] pair = maxpq.poll();
        // maxval 是 nums2 中的最大值，i 是对应索引
        int i = pair[0], maxval = pair[1];
        if (maxval < nums1[right]) {
            // 如果 nums1[right] 能胜过 maxval，那就自己上
            res[i] = nums1[right];
            right--;
        } else {
            // 否则用最小值混一下，养精蓄锐
            res[i] = nums1[left];
            left++;
        }
    }
    return res;
}
```

算法的时间复杂度很好分析，也就是二叉堆和排序的复杂度 O(nlogn)。

至此，这道田忌赛马的题就解决了，其代码实现上用到了双指针技巧，从最快的马开始，比得过就比，比不过就送，这样就能对任意数量的马求取一个最优的比赛策略了。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



一文秒杀四道原地修改数组的算法题

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

26. 删除有序数组中的重复项（简单）

83. 删除排序链表中的重复元素（简单）

27. 移除元素（简单）

283. 移动零（简单）

我们知道对于数组来说，在尾部插入、删除元素是比较高效的，时间复杂度是 O(1)，但是如果在中间或者开头插入、删除元素，就会涉及数据的搬移，时间复杂度为 O(N)，效率较低。

所以上篇文章 常数时间删除/查找数组中的任意元素 就讲了一种技巧，把待删除元素交换到最后一个，然后再删除，就可以避免数据搬移。

那么这篇文章我们换一个场景，来讲一讲如何在原地修改数组，避免数据的搬移。

有序数组/链表去重

先讲讲如何对一个有序数组去重，先看下力扣第 26 题「删除有序数组中的重复项」：

26. 删除排序数组中的重复项 labuladong 题解 思路

难度 简单 1658

给定一个排序数组，你需要在 原地 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 原地 **修改输入数组** 并在使用 O(1) 额外空间的条件下完成。

示例 1:

```
给定数组 nums = [1,1,2],

函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。

你不需要考虑数组中超出新长度后面的元素。
```

示例 2:

```
给定 nums = [0,0,1,1,1,2,2,3,3,4],

函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。

你不需要考虑数组中超出新长度后面的元素。
```

函数签名如下：

```
int removeDuplicates(int[] nums);
```

显然，由于数组已经排序，所以重复的元素一定连在一起，找出它们并不难，但如果每找到一个重复元素就立即删除它，就是在数组中间进行删除操作，整个时间复杂度是会达到 O(N^2)。

简单解释一下什么是原地修改：

如果不是原地修改的话，我们直接 new 一个 `int[]` 数组，把去重之后的元素放进这个新数组中，然后返回这个新数组即可。

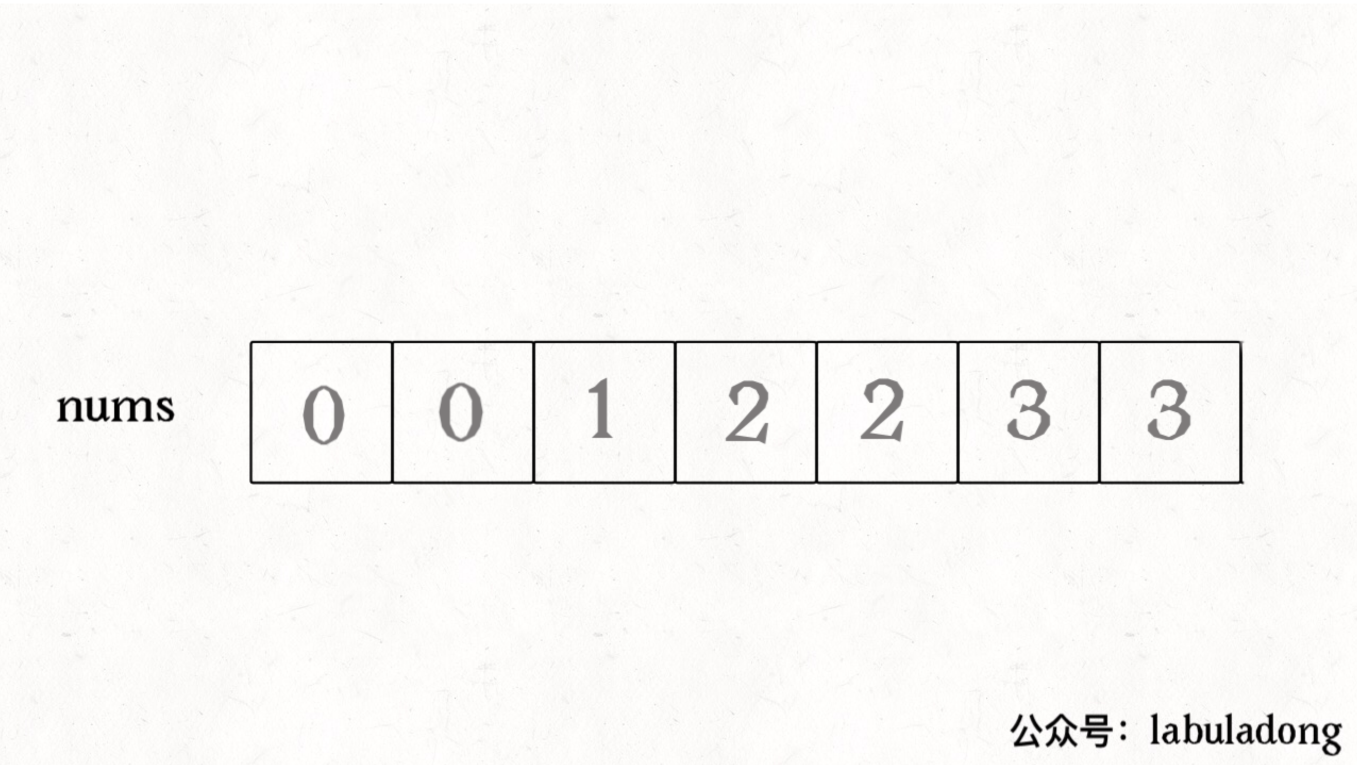
但是原地删除，不允许我们 new 新数组，只能在原数组上操作，然后返回一个长度，这样就可以通过返回的长度和原始数组得到我们去重后的元素有哪些了。

这种需求在数组相关的算法题中时非常常见的，**通用解法就是我们前文 双指针技巧 中的快慢指针技巧。**

我们让慢指针 `slow` 走在后面，快指针 `fast` 走在前面探路，找到一个不重复的元素就告诉 `slow` 并让 `slow` 前进一步。这样当 `fast` 指针遍历完整个数组 `nums` 后，**`nums[0..slow]` 就是不重复元素。**

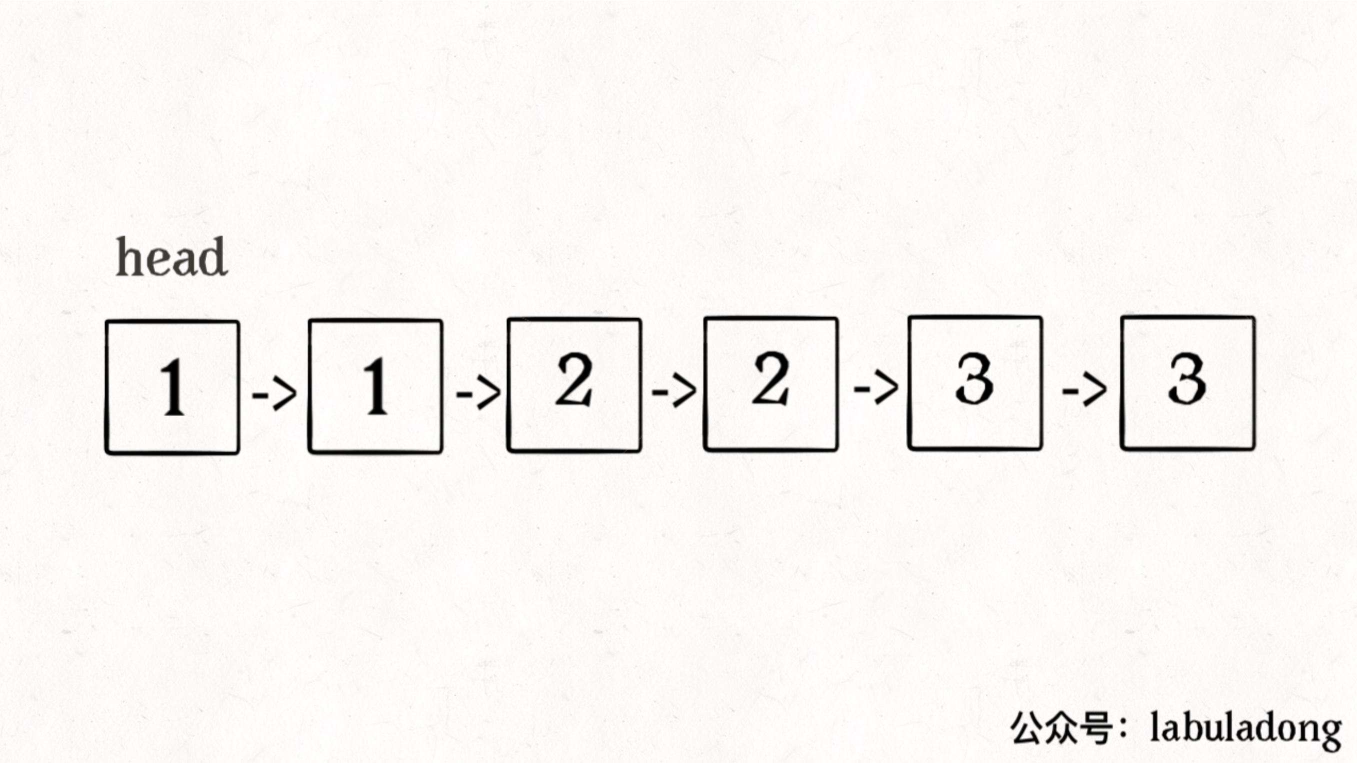
```
int removeDuplicates(int[] nums) {
    if (nums.length == 0) {
        return 0;
    }
    int slow = 0, fast = 0;
    while (fast < nums.length) {
        if (nums[fast] != nums[slow]) {
            slow++;
            // 维护 nums[0..slow] 无重复
            nums[slow] = nums[fast];
        }
        fast++;
    }
    // 数组长度为索引 + 1
    return slow + 1;
}
```

看下算法执行的过程：



再简单扩展一下，如果给你一个有序链表，如何去重呢？这是力扣第 83 题「删除排序链表中的重复元素」，其实和数组去重是一模一样的，唯一的区别是把数组赋值操作变成操作指针而已：

```java
ListNode deleteDuplicates(ListNode head) {
    if (head == null) return null;
    ListNode slow = head, fast = head;
    while (fast != null) {
        if (fast.val != slow.val) {
            // nums[slow] = nums[fast];
            slow.next = fast;
            // slow++;
            slow = slow.next;
        }
        // fast++
        fast = fast.next;
    }
    // 断开与后面重复元素的连接
    slow.next = null;
    return head;
}
```



移除元素

这是力扣第 27 题「移除元素」，看下题目：

27. 移除元素 labuladong 题解 思路

难度 简单 667

给你一个数组 *nums* 和一个值 *val*，你需要 原地 移除所有数值等于 *val* 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并 原地 **修改输入数组**。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

示例 1:

```
给定 nums = [3,2,2,3], val = 3,

函数应该返回新的长度 2, 并且 nums 中的前两个元素均为 2。

你不需要考虑数组中超出新长度后面的元素。
```

示例 2:

```
给定 nums = [0,1,2,2,3,0,4,2], val = 2,

函数应该返回新的长度 5, 并且 nums 中的前五个元素为 0, 1, 3, 0, 4。

注意这五个元素可为任意顺序。

你不需要考虑数组中超出新长度后面的元素。
```

函数签名如下：

```java
int removeElement(int[] nums, int val);
```

题目要求我们把 `nums` 中所有值为 `val` 的元素原地删除，依然需要使用 双指针技巧 中的快慢指针：

如果 `fast` 遇到需要去除的元素，则直接跳过，否则就告诉 `slow` 指针，并让 `slow` 前进一步。

这和前面说到的数组去重问题解法思路是完全一样的，就不画 GIF 了，直接看代码：

```java
int removeElement(int[] nums, int val) {
    int fast = 0, slow = 0;
    while (fast < nums.length) {
        if (nums[fast] != val) {
            nums[slow] = nums[fast];
            slow++;
        }
        fast++;
    }
    return slow;
}
```

注意这里和有序数组去重的解法有一个重要不同，我们这里是先给 `nums[slow]` 赋值然后再给 `slow++`，这样可以保证 `nums[0..slow-1]` 是不包含值为 `val` 的元素的，最后的结果数组长度就是 `slow`。

移动零

这是力扣第 283 题「移动零」，我来描述下题目：

给你输入一个数组 `nums`，请你**原地修改**，将数组中的所有值为 0 的元素移到数组末尾，函数签名如下：

```java
void moveZeroes(int[] nums);
```

比如说给你输入 `nums = [0,1,4,0,2]`，你的算法没有返回值，但是会把 `nums` 数组原地修改成 `[1,4,2,0,0]`。

结合之前说到的几个题目，你是否有已经有了答案呢？

题目让我们将所有 0 移到最后，其实就相当于移除 `nums` 中的所有 0，然后再把后面的元素都赋值为 0 即可。

所以我们可以复用上一题的 `removeElement` 函数：

```java
void moveZeroes(int[] nums) {
    // 去除 nums 中的所有 0
    // 返回去除 0 之后的数组长度
    int p = removeElement(nums, 0);
    // 将 p 之后的所有元素赋值为 0
    for (; p < nums.length; p++) {
        nums[p] = 0;
    }
}

// 见上文代码实现
int removeElement(int[] nums, int val);
```

至此，四道「原地修改」的算法问题就讲完了，其实核心还是快慢指针技巧，你学会了吗？

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

——————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



一文搞懂单链表的六大解题套路

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

21. 合并两个有序链表（简单）

23. 合并K个升序链表（困难）

141. 环形链表（简单）

142. 环形链表 II（中等）

876. 链表的中间结点（简单）

19. 删除链表的倒数第 N 个结点（中等）

160. 相交链表（简单）

上次在视频号直播，跟大家说到单链表有很多巧妙的操作，本文就总结一下单链表的基本技巧，每个技巧都对应着至少一道算法题：

1、合并两个有序链表

2、合并 `k` 个有序链表

3、寻找单链表的倒数第 `k` 个节点

4、寻找单链表的中点

5、判断单链表是否包含环并找出环起点

6、判断两个单链表是否相交并找出交点

这些解法都用到了双指针技巧，所以说对于单链表相关的题目，双指针的运用是非常广泛的，下面我们就来一个一个看。

合并两个有序链表

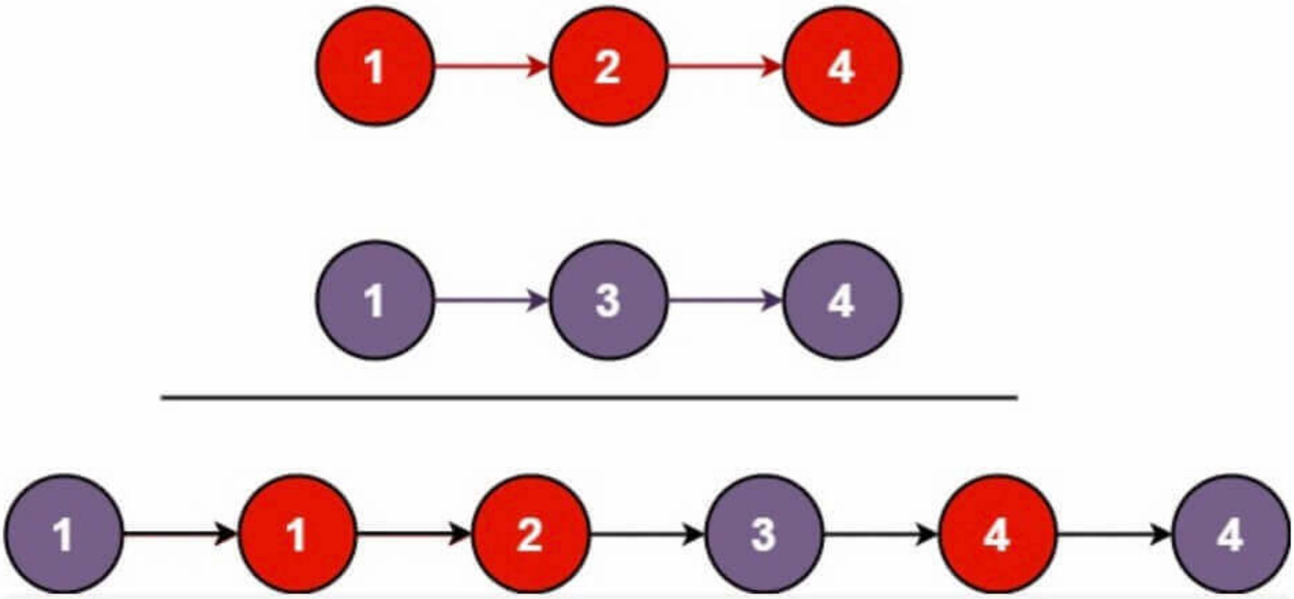
这是最基本的链表技巧，力扣第 21 题「合并两个有序链表」就是这个问题：

21. 合并两个有序链表 labuladong 题解 思路

难度 简单 1846

将两个升序链表合并为一个新的 **升序** 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1：




```
输入：l1 = [1,2,4], l2 = [1,3,4]
输出：[1,1,2,3,4,4]
```

给你输入两个有序链表，请你把他俩合并成一个新的有序链表，函数签名如下：

```
ListNode mergeTwoLists(ListNode l1, ListNode l2);
```

这题比较简单，我们直接看解法：

```
ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    // 虚拟头结点
    ListNode dummy = new ListNode(-1), p = dummy;
    ListNode p1 = l1, p2 = l2;

    while (p1 != null && p2 != null) {
        // 比较 p1 和 p2 两个指针
        // 将值较小的的节点接到 p 指针
```

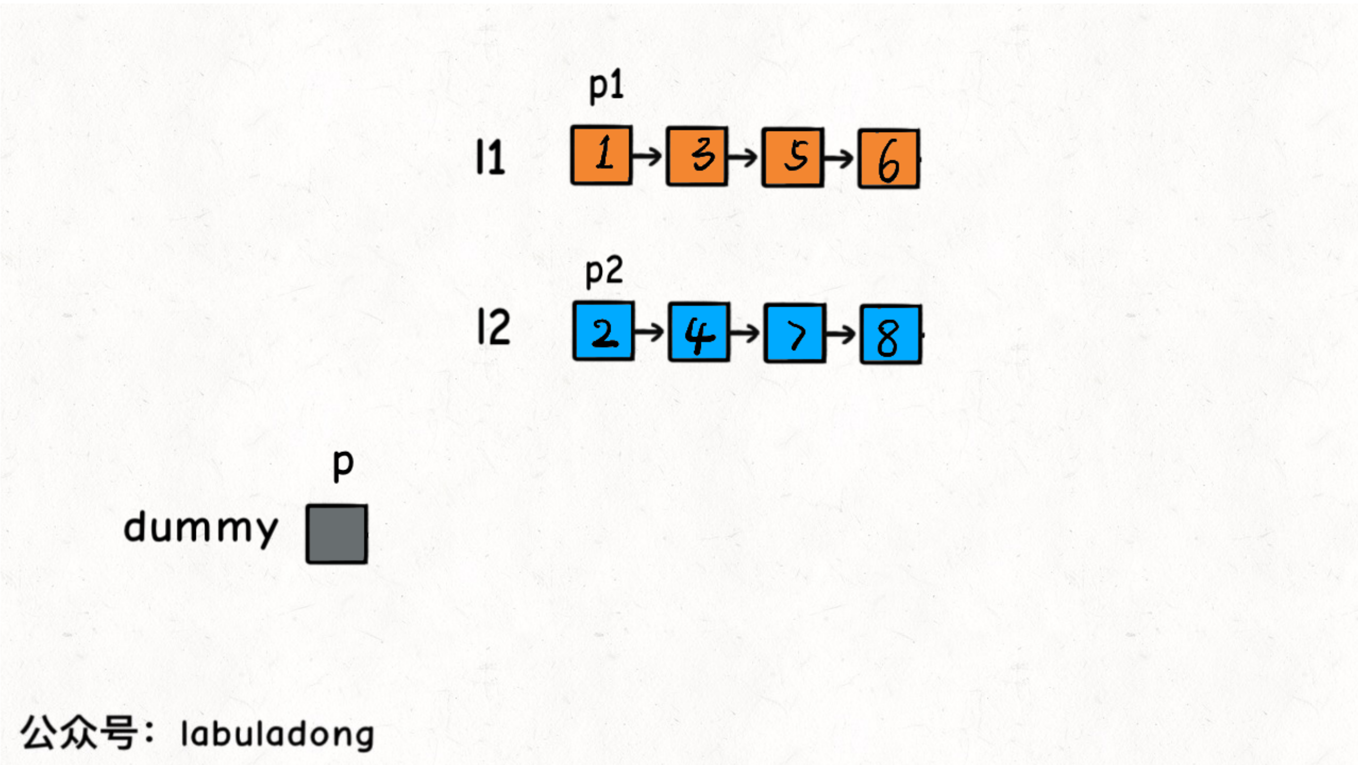
```
        if (p1.val > p2.val) {
            p.next = p2;
            p2 = p2.next;
        } else {
            p.next = p1;
            p1 = p1.next;
        }
        // p 指针不断前进
        p = p.next;
    }

    if (p1 != null) {
        p.next = p1;
    }

    if (p2 != null) {
        p.next = p2;
    }

    return dummy.next;
}
```

我们的 while 循环每次比较 p1 和 p2 的大小，把较小的节点接到结果链表上：

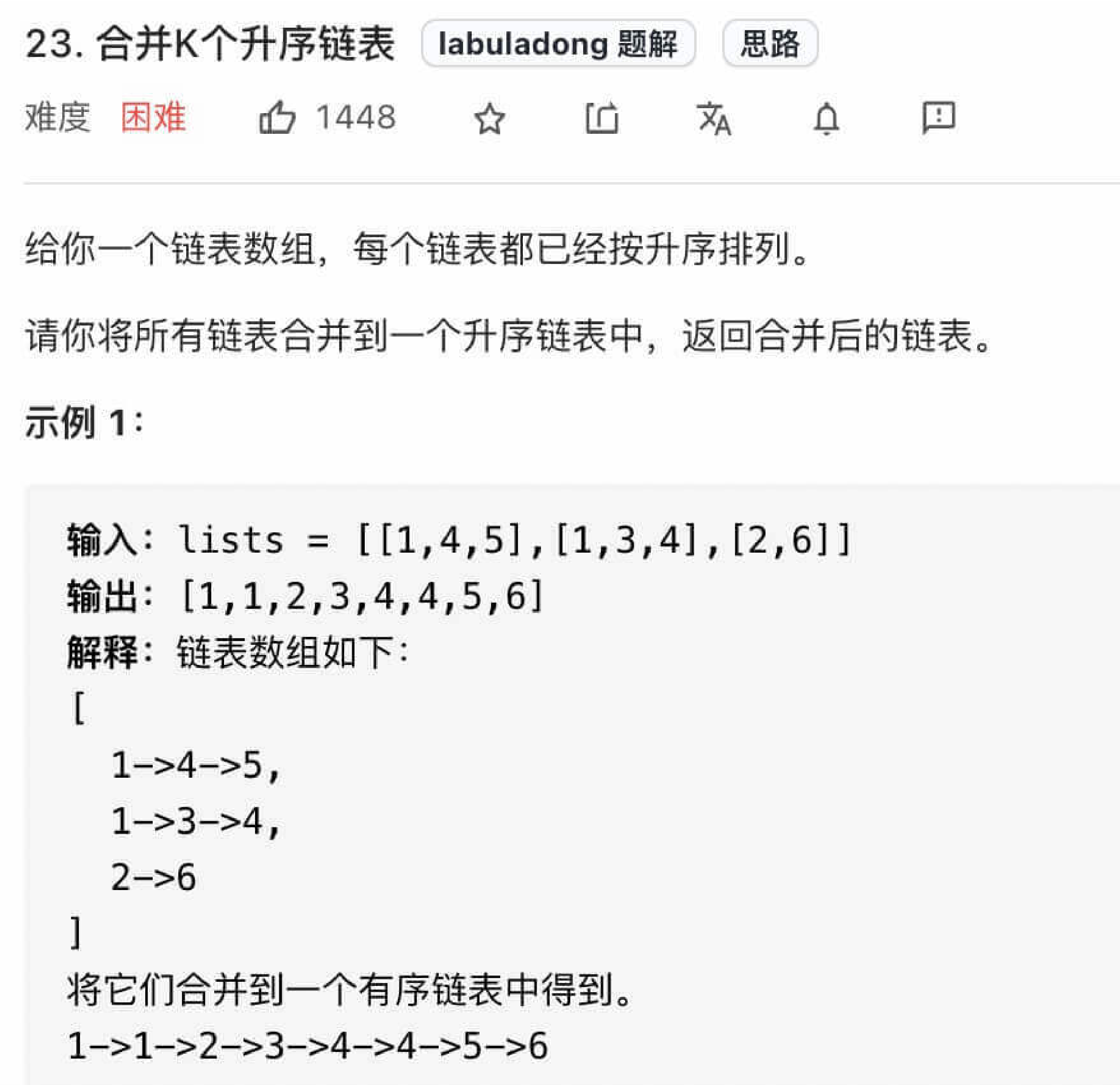


这个算法的逻辑类似于「拉拉链」，l1, l2 类似于拉链两侧的锯齿，指针 p 就好像拉链的拉索，将两个有序链表合并。

代码中还用到一个链表的算法题中是很常见的「虚拟头结点」技巧，也就是 dummy 节点。你可以试试，如果不使用 dummy 虚拟节点，代码会复杂很多，而有了 dummy 节点这个占位符，可以避免处理空指针的情况，降低代码的复杂性。

合并 k 个有序链表

看下力扣第 23 题「合并K个升序链表」：



23. 合并K个升序链表 labuladong 题解 思路

难度 困难 1448

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

```
输入：lists = [[1,4,5],[1,3,4],[2,6]]
输出：[1,1,2,3,4,4,5,6]
解释：链表数组如下：
[
  1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6
```

函数签名如下：

```java
ListNode mergeKLists(ListNode[] lists);
```

合并 `k` 个有序链表的逻辑类似合并两个有序链表，难点在于，如何快速得到 `k` 个节点中的最小节点，接到结果链表上？

这里我们就要用到 优先级队列（二叉堆） 这种数据结构，把链表节点放入一个最小堆，就可以每次获得 `k` 个节点中的最小节点：

```java
ListNode mergeKLists(ListNode[] lists) {
    if (lists.length == 0) return null;
    // 虚拟头结点
    ListNode dummy = new ListNode(-1);
    ListNode p = dummy;
    // 优先级队列，最小堆
    PriorityQueue<ListNode> pq = new PriorityQueue<>(
        lists.length, (a, b)->(a.val - b.val));
    // 将 k 个链表的头结点加入最小堆
    for (ListNode head : lists) {
        if (head != null)
            pq.add(head);
    }

    while (!pq.isEmpty()) {
        // 获取最小节点，接到结果链表中
        ListNode node = pq.poll();
        p.next = node;
        if (node.next != null) {
            pq.add(node.next);
        }
        // p 指针不断前进
        p = p.next;
    }
    return dummy.next;
}
```

这个算法是面试常考题，它的时间复杂度是多少呢？

优先队列 `pq` 中的元素个数最多是 `k`，所以一次 `poll` 或者 `add` 方法的时间复杂度是 `O(logk)`；所有的链表节点都会被加入和弹出 `pq`，**所以算法整体的时间复杂度是 `O(Nlogk)`，其中 `k` 是链表的条数，`N` 是这些链表的节点总数**。

单链表的倒数第 k 个节点

从前往后寻找单链表的第 `k` 个节点很简单，一个 for 循环遍历过去就找到了，但是如何寻找从后往前数的第 `k` 个节点呢？

那你可能说，假设链表有 `n` 个节点，倒数第 `k` 个节点就是正数第 `n - k` 个节点，不也是一个 for 循环的事儿吗？

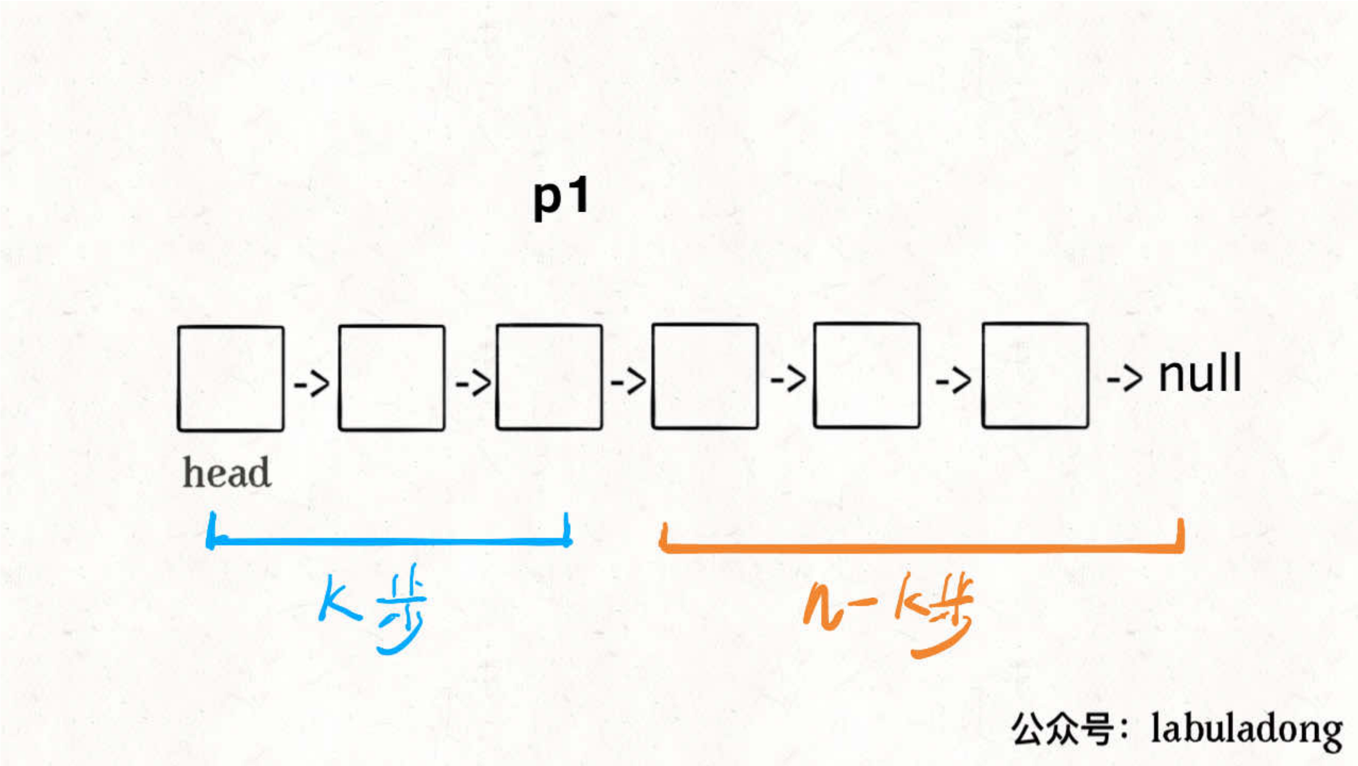
是的，但是算法题一般只给你一个 `ListNode` 头结点代表一条单链表，你不能直接得出这条链表的长度 `n`，而需要先遍历一遍链表算出 `n` 的值，然后再遍历链表计算第 `n - k` 个节点。

也就是说，这个解法需要遍历两次链表才能得到出倒数第 `k` 个节点。

那么，我们能不能**只遍历一次链表**，就算出倒数第 `k` 个节点？可以做到的，如果是面试问到这道题，面试官肯定也是希望你给出只需遍历一次链表的解法。

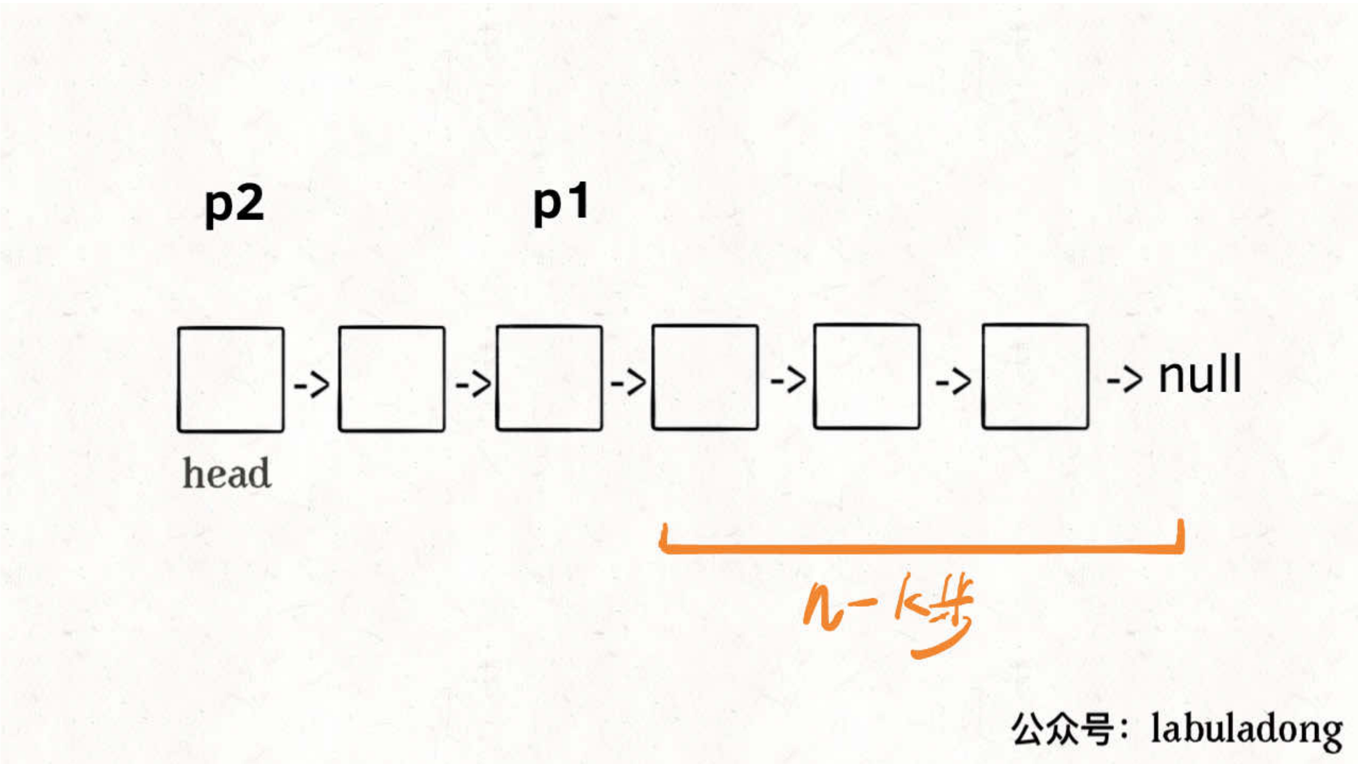
这个解法就比较巧妙了，假设 `k = 2`，思路如下：

首先，我们先让一个指针 p1 指向链表的头节点 head，然后走 k 步：

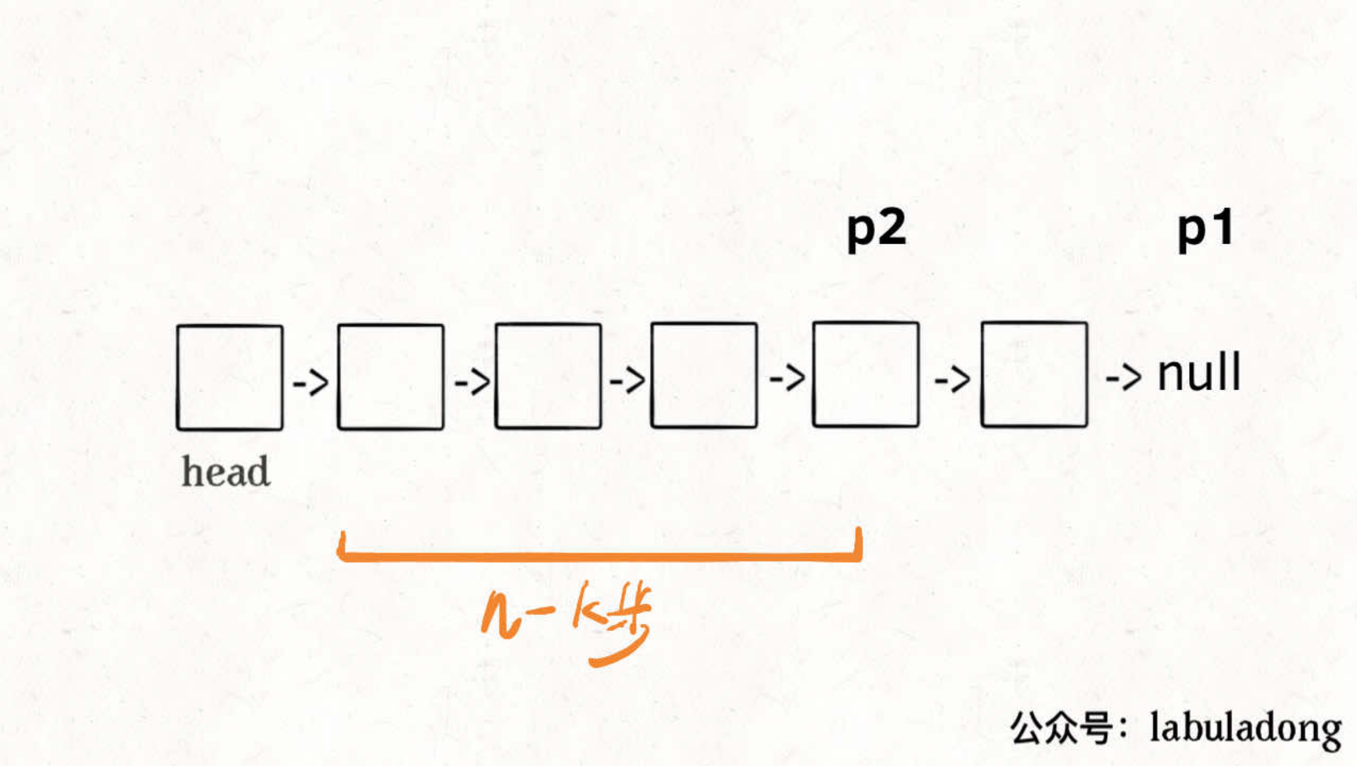


现在的 p1，只要再走 n - k 步，就能走到链表末尾的空指针了对吧？

趁这个时候，再用一个指针 p2 指向链表头节点 head：



接下来就很显然了，让 p1 和 p2 同时向前走，p1 走到链表末尾的空指针时走了 n - k 步，p2 也走了 n - k 步，也就是链表的倒数第 k 个节点：



这样，只遍历了一次链表，就获得了倒数第 k 个节点 p2。

上述逻辑的代码如下：

```
// 返回链表的倒数第 k 个节点
ListNode findFromEnd(ListNode head, int k) {
    ListNode p1 = head;
    // p1 先走 k 步
    for (int i = 0; i < k; i++) {
        p1 = p1.next;
    }
    ListNode p2 = head;
    // p1 和 p2 同时走 n - k 步
    while (p1 != null) {
        p2 = p2.next;
        p1 = p1.next;
    }
    // p2 现在指向第 n - k 个节点
    return p2;
}
```

当然，如果用 big O 表示法来计算时间复杂度，无论遍历一次链表和遍历两次链表的时间复杂度都是 O(N)，但上述这个算法更有技巧性。

很多链表相关的算法题都会用到这个技巧，比如说力扣第 19 题「删除链表的倒数第 N 个结点」：

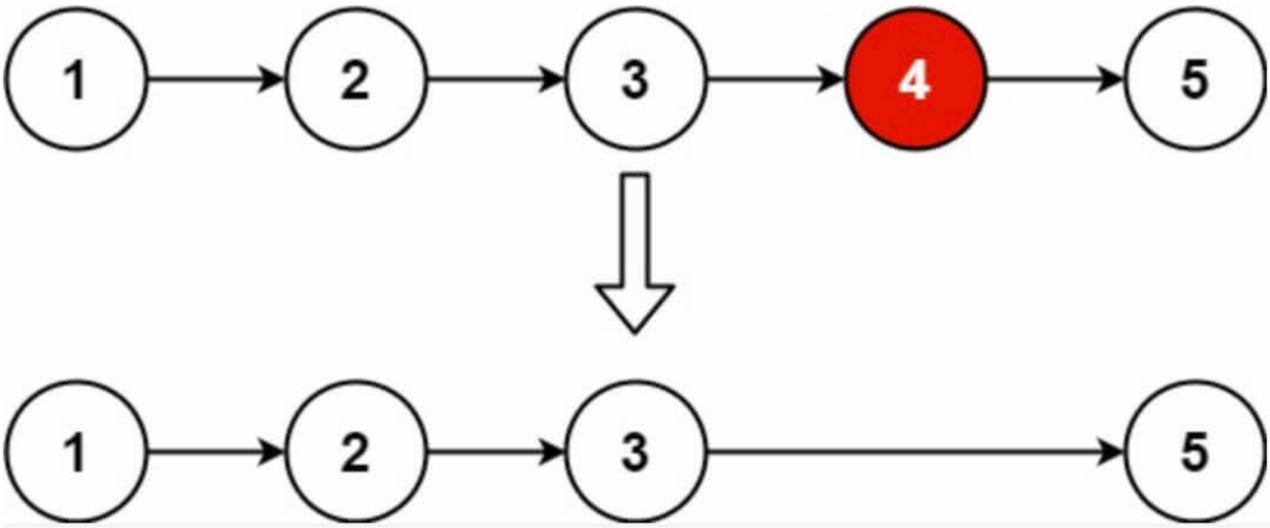
19. 删除链表的倒数第 N 个结点 labuladong 题解 思路

难度 中等 1506 收藏 分享 切换为英文 接收动态 反馈

给你一个链表，删除链表的倒数第 `n` 个结点，并且返回链表的头结点。

进阶： 你能尝试使用一趟扫描实现吗？

示例 1：




```
输入：head = [1,2,3,4,5], n = 2
输出：[1,2,3,5]
```

我们直接看解法代码：

```java
// 主函数
public ListNode removeNthFromEnd(ListNode head, int n) {
    // 虚拟头结点
    ListNode dummy = new ListNode(-1);
    dummy.next = head;
    // 删除倒数第 n 个，要先找倒数第 n + 1 个节点
    ListNode x = findFromEnd(dummy, n + 1);
    // 删掉倒数第 n 个节点
    x.next = x.next.next;
    return dummy.next;
}

private ListNode findFromEnd(ListNode head, int k) {
    // 代码见上文
}
```

这个逻辑就很简单了，要删除倒数第 `n` 个节点，就得获得倒数第 `n + 1` 个节点的引用，可以用我们实现的 `findFromEnd` 来操作。

不过注意我们又使用了虚拟头结点的技巧，也是为了防止出现空指针的情况，比如说链表总共有 5 个节点，题目就让你删除倒数第 5 个节点，也就是第一个节点，那按照算法逻辑，应该首先找到倒数第 6 个节点。但第一个节点前面已经没有节点了，这就会出错。

但有了我们虚拟节点 `dummy` 的存在，就避免了这个问题，能够对这种情况进行正确的删除。

单链表的中点

这个技巧在前文 双指针技巧汇总 写过，如果看过的读者可以跳过。

力扣第 876 题「链表的中间结点」就是这个题目，问题的关键也在于我们无法直接得到单链表的长度 `n`，常规方法也是先遍历链表计算 `n`，再遍历一次得到第 `n / 2` 个节点，也就是中间节点。

如果想一次遍历就得到中间节点，也需要耍点小聪明，使用「快慢指针」的技巧：

我们让两个指针 `slow` 和 `fast` 分别指向链表头结点 `head`。

每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步，这样，当 `fast` 走到链表末尾时，`slow` 就指向了链表中点。

上述思路的代码实现如下：

```
ListNode middleNode(ListNode head) {
    // 快慢指针初始化指向 head
    ListNode slow = head, fast = head;
    // 快指针走到末尾时停止
    while (fast != null && fast.next != null) {
        // 慢指针走一步，快指针走两步
        slow = slow.next;
        fast = fast.next.next;
    }
    // 慢指针指向中点
    return slow;
}
```

需要注意的是，如果链表长度为偶数，也就是说中点有两个的时候，我们这个解法返回的节点是靠后的那个节点。

另外，这段代码稍加修改就可以直接用到判断链表成环的算法题上。

判断链表是否包含环

这个技巧也在前文 双指针技巧汇总 写过，如果看过的读者可以跳过。

判断链表是否包含环属于经典问题了，解决方案也是用快慢指针：

每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步。

如果 `fast` 最终遇到空指针，说明链表中没有环；如果 `fast` 最终和 `slow` 相遇，那肯定是 `fast` 超过了 `slow` 一圈，说明链表中含有环。

只需要把寻找链表中点的代码稍加修改就行了：

```
boolean hasCycle(ListNode head) {
    // 快慢指针初始化指向 head
    ListNode slow = head, fast = head;
    // 快指针走到末尾时停止
    while (fast != null && fast.next != null) {
        // 慢指针走一步，快指针走两步
        slow = slow.next;
        fast = fast.next.next;
        // 快慢指针相遇，说明含有环
        if (slow == fast) {
            return true;
        }
    }
    // 不包含环
    return false;
}
```

当然，这个问题还有进阶版：如果链表中含有环，如何计算这个环的起点？

这里简单提一下解法：

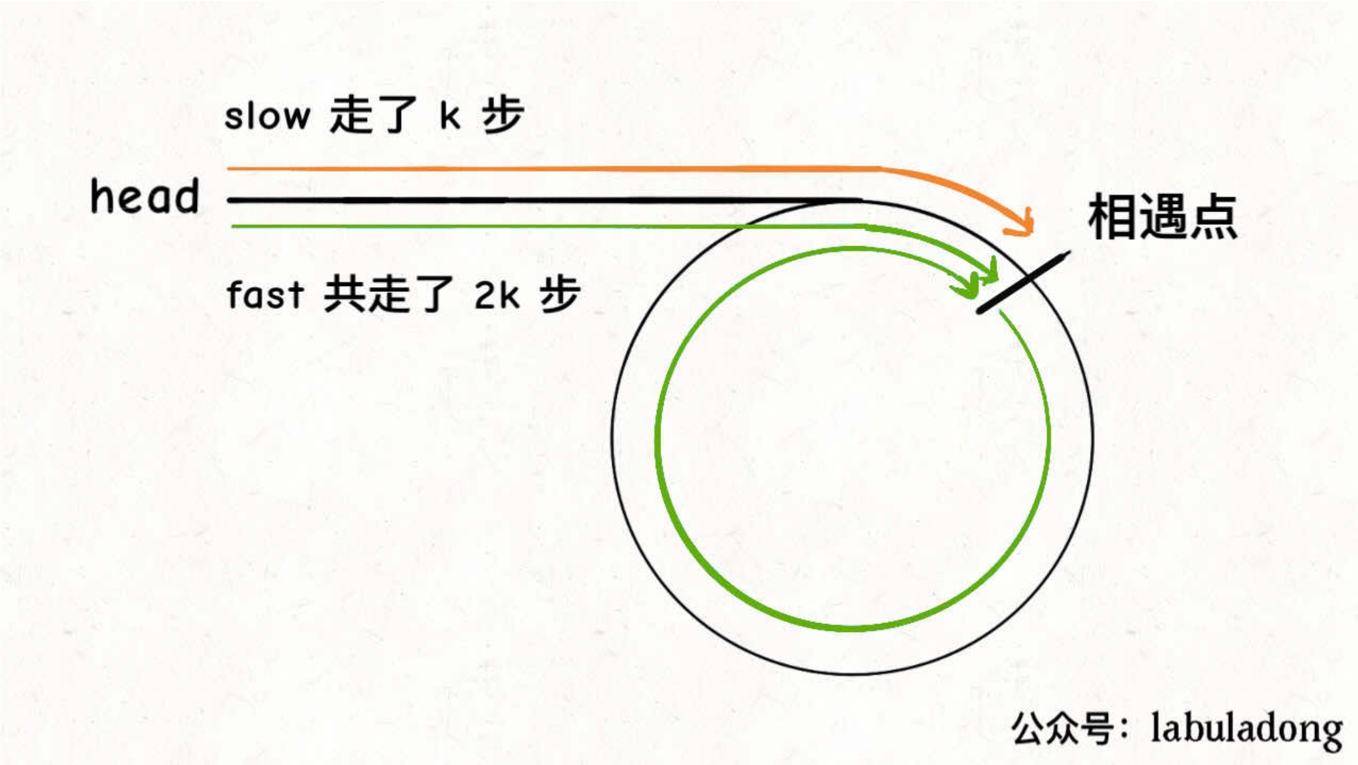
```
ListNode detectCycle(ListNode head) {
    ListNode fast, slow;
    fast = slow = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
        if (fast == slow) break;
    }
    // 上面的代码类似 hasCycle 函数
    if (fast == null || fast.next == null) {
        // fast 遇到空指针说明没有环
        return null;
    }

    // 重新指向头结点
    slow = head;
    // 快慢指针同步前进，相交点就是环起点
    while (slow != fast) {
        fast = fast.next;
        slow = slow.next;
    }
    return slow;
}
```

可以看到，当快慢指针相遇时，让其中任一个指针指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置。

前文 双指针技巧汇总 详细解释了其中的原理，这里简单说一下。

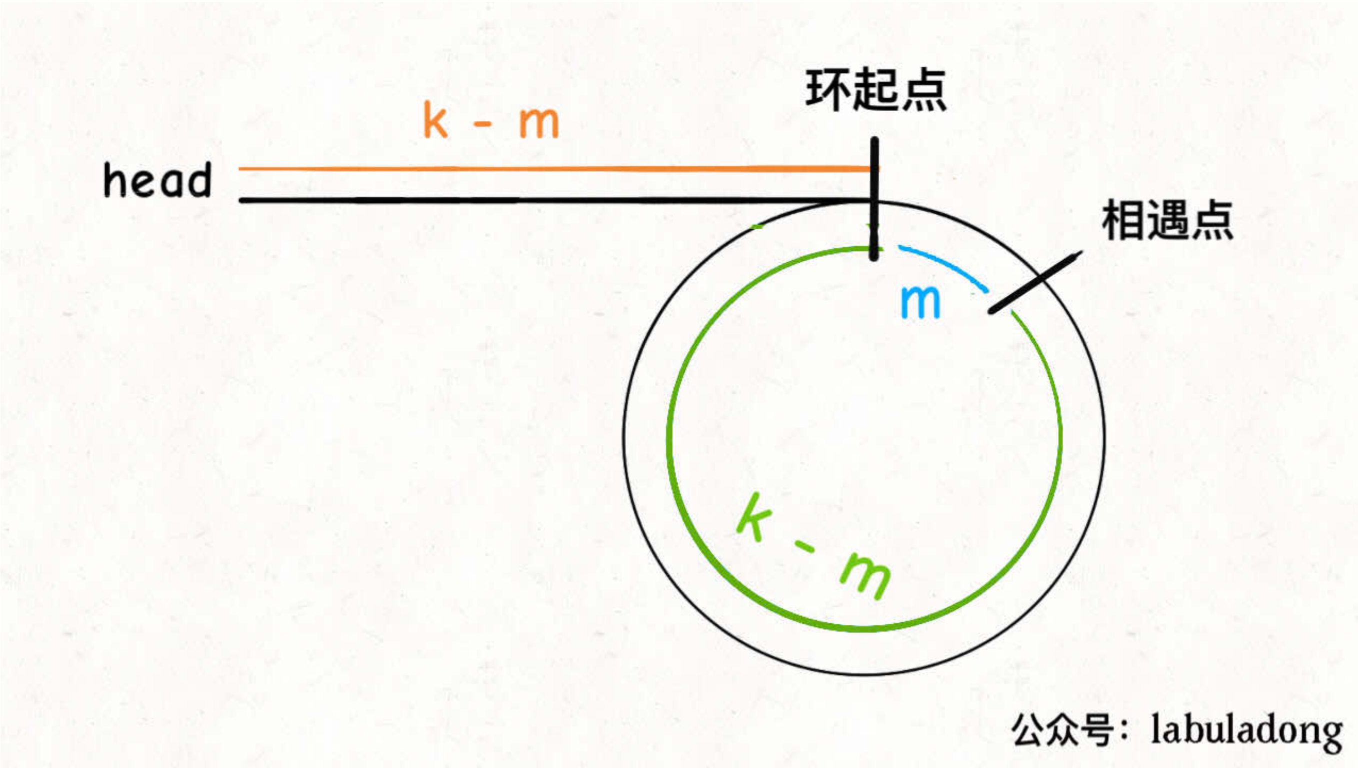
我们假设快慢指针相遇时，慢指针 `slow` 走了 `k` 步，那么快指针 `fast` 一定走了 `2k` 步：



`fast` 一定比 `slow` 多走了 `k` 步，这多走的 `k` 步其实就是 `fast` 指针在环里转圈圈，所以 `k` 的值就是环长度的「整数倍」。

假设相遇点距环的起点的距离为 `m`，那么结合上图的 `slow` 指针，环的起点距头结点 `head` 的距离为 `k - m`，也就是说如果从 `head` 前进 `k - m` 步就能到达环起点。

巧的是，如果从相遇点继续前进 `k - m` 步，也恰好到达环起点。因为结合上图的 `fast` 指针，从相遇点开始走k步可以转回到相遇点，那走 `k - m` 步肯定就走到环起点了：



所以，只要我们把快慢指针中的任一个重新指向 `head`，然后两个指针同速前进，`k - m` 步后一定会相遇，相遇之处就是环的起点了。

两个链表是否相交

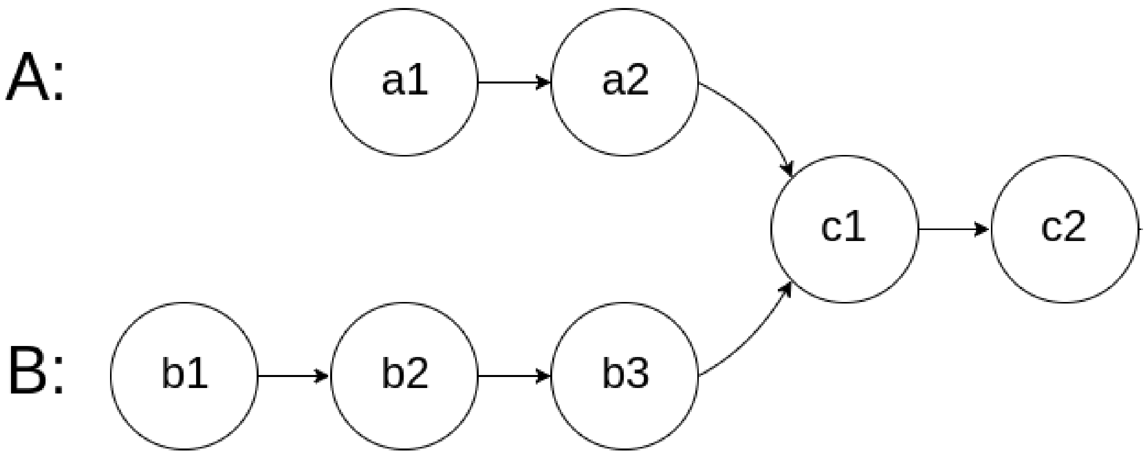
这个问题有意思，也是力扣第 160 题「[相交链表](#)」函数签名如下：

```
ListNode getIntersectionNode(ListNode headA, ListNode headB);
```

给你输入两个链表的头结点 `headA` 和 `headB`，这两个链表可能存在相交。

如果相交，你的算法应该返回相交的那个节点；如果没相交，则返回 null。

比如题目给我们举的例子，如果输入的两个链表如下图：

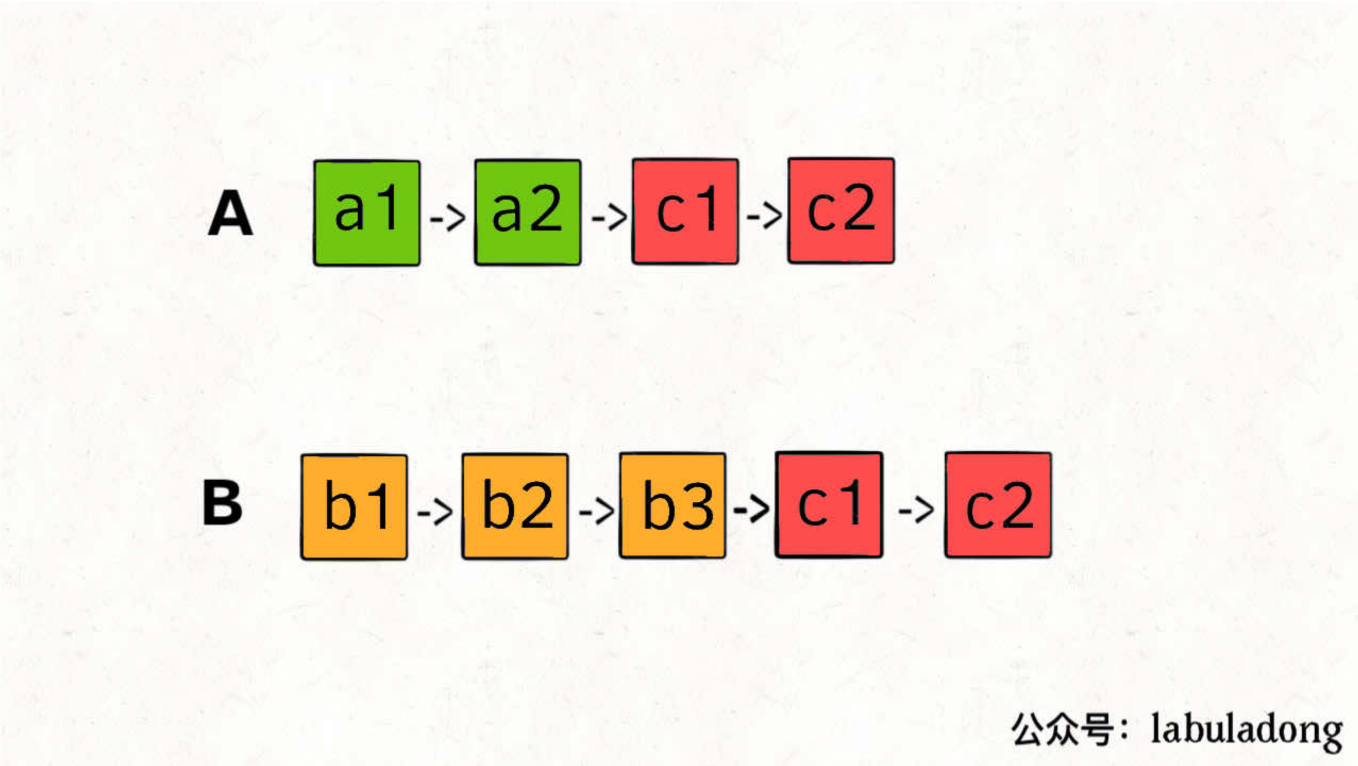


那么我们的算法应该返回 `c1` 这个节点。

这个题直接的想法可能是用 `HashSet` 记录一个链表的所有节点，然后和另一条链表对比，但这就需要额外的空间。

如果不用额外的空间，只使用两个指针，你如何做呢？

难点在于，由于两条链表的长度可能不同，两条链表之间的节点无法对应：

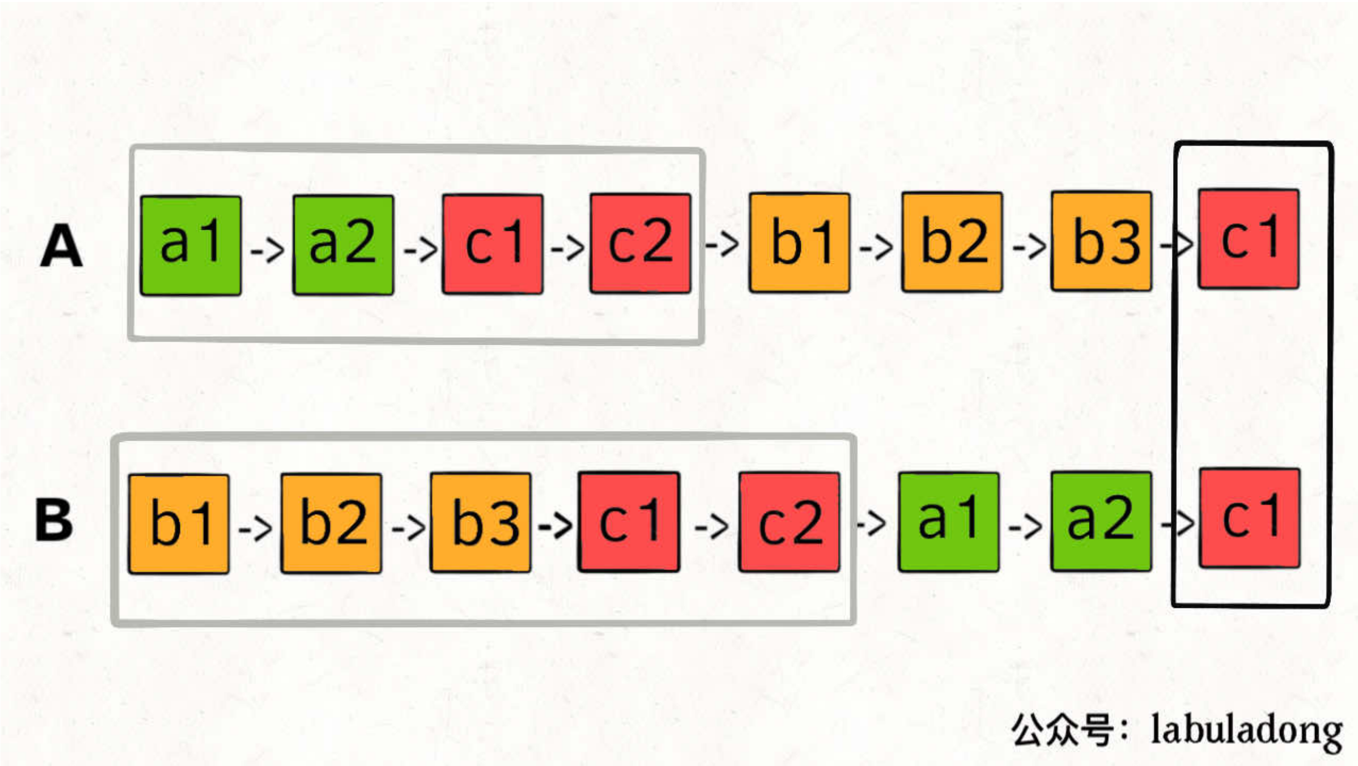


如果用两个指针 `p1` 和 `p2` 分别在两条链表上前进，并不能**同时**走到公共节点，也就无法得到相交节点 `c1`。

解决这个问题的关键是，通过某些方式，让 `p1` 和 `p2` 能够同时到达相交节点 `c1`。

所以，我们可以让 `p1` 遍历完链表 `A` 之后开始遍历链表 `B`，让 `p2` 遍历完链表 `B` 之后开始遍历链表 `A`，这样相当于「逻辑上」两条链表接在了一起。

如果这样进行拼接，就可以让 `p1` 和 `p2` 同时进入公共部分，也就是同时到达相交节点 `c1`：



那你可能会问，如果说两个链表没有相交点，是否能够正确的返回 null 呢？

这个逻辑可以覆盖这种情况的，相当于 `c1` 节点是 null 空指针嘛，可以正确返回 null。

按照这个思路，可以写出如下代码：

```
ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    // p1 指向 A 链表头结点，p2 指向 B 链表头结点
    ListNode p1 = headA, p2 = headB;
    while (p1 != p2) {
        // p1 走一步，如果走到 A 链表末尾，转到 B 链表
        if (p1 == null) p1 = headB;
        else            p1 = p1.next;
        // p2 走一步，如果走到 B 链表末尾，转到 A 链表
        if (p2 == null) p2 = headA;
        else            p2 = p2.next;
    }
    return p1;
}
```

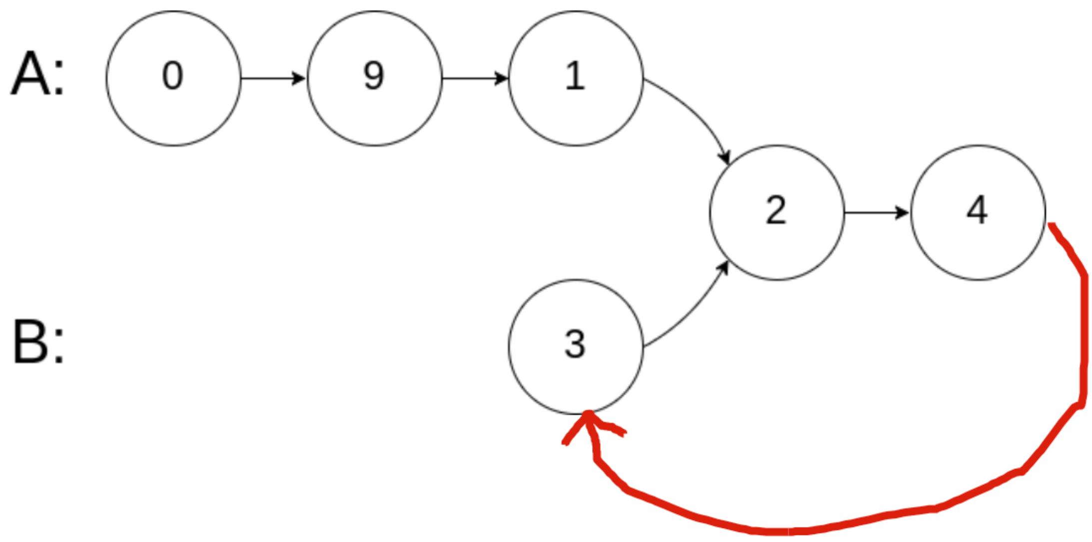
这样，这道题就解决了，空间复杂度为 `O(1)`，时间复杂度为 `O(N)`。

以上就是单链表的所有技巧，希望对你有启发。

2022/1/24 更新：

评论区有不少优秀读者对最后一题「寻找两条链表的交点」提出了一些其他思路，也补充到这里。

首先有读者提到，如果把两条链表首尾相连，那么「寻找两条链表的交点」的问题转换成了前面讲的「寻找环起点」的问题：



说实话我没有想到这种思路，不得不说这是一个很巧妙的转换！

另外，还有读者提到，既然「寻找两条链表的交点」的核心在于让 p1 和 p2 两个指针能够同时到达相交节点 c1，那么可以通过预先计算两条链表的长度来做到这一点，具体代码如下：

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    int lenA = 0, lenB = 0;
    // 计算两条链表的长度
    for (ListNode p1 = headA; p1 != null; p1 = p1.next) {
        lenA++;
    }
    for (ListNode p2 = headB; p2 != null; p2 = p2.next) {
        lenB++;
    }
    // 让 p1 和 p2 到达尾部的距离相同
    ListNode p1 = headA, p2 = headB;
    if (lenA > lenB) {
        for (int i = 0; i < lenA - lenB; i++) {
            p1 = p1.next;
        }
    } else {
        for (int i = 0; i < lenB - lenA; i++) {
            p2 = p2.next;
        }
    }
    // 看两个指针是否会相同，p1 == p2 时有两种情况：
    // 1、要么是两条链表不相交，他俩同时走到尾部空指针
    // 2、要么是两条链表相交，他俩走到两条链表的相交点
    while (p1 != p2) {
        p1 = p1.next;
        p2 = p2.next;
```

```
        }
        return p1;
    }
```

虽然代码多一些，但是时间复杂度是还是 O(N)，而且会更容易理解一些。

总之，我的解法代码并不一定就是最优或者最正确的，鼓励大家在评论区多多提出自己的疑问和思考，我也很高兴和大家探讨更多的解题思路~

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

＿＿＿＿＿＿＿＿＿＿＿＿＿

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



链表操作的递归思维一览

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

206. 反转链表（简单）

92. 反转链表 II（中等）

反转单链表的迭代实现不是一个困难的事情，但是递归实现就有点难度了，如果再加一点难度，让你仅仅反转单链表中的一部分，你是否能**够递归实现**呢？

本文就来由浅入深，step by step 地解决这个问题。如果你还不会递归地反转单链表也没关系，**本文会从递归反转整个单链表开始拓展**，只要你明白单链表的结构，相信你能够有所收获。

```
// 单链表节点的结构
public class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}
```

什么叫反转单链表的一部分呢，就是给你一个索引区间，让你把单链表中这部分元素反转，其他部分不变。

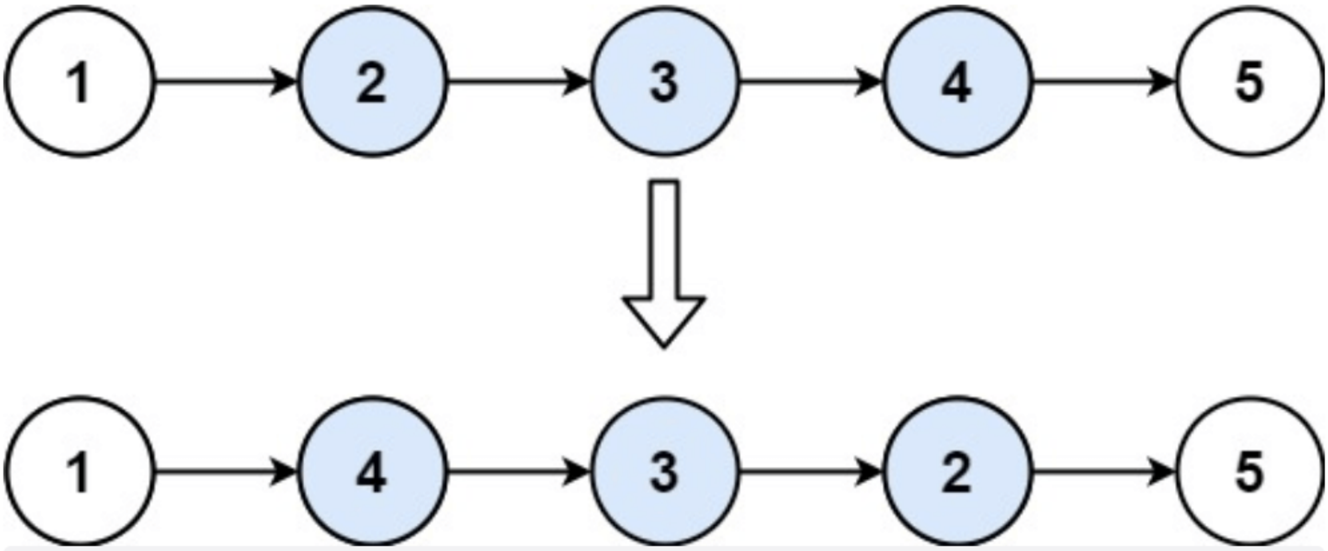
看下力扣第 92 题「反转链表 II」：

92. 反转链表 II labuladong 题解 思路

难度 中等 1162

给你单链表的头指针 `head` 和两个整数 `left` 和 `right` ，其中 `left <= right` 。请你反转从位置 `left` 到位置 `right` 的链表节点，返回 **反转后的链表** 。

示例 1：




```
输入：head = [1,2,3,4,5], left = 2, right = 4
输出：[1,4,3,2,5]
```

注意这里的索引是从 1 开始的。迭代的思路大概是：先用一个 for 循环找到第 `m` 个位置，然后再用一个 for 循环将 `m` 和 `n` 之间的元素反转。但是我们的递归解法不用一个 for 循环，纯递归实现反转。

迭代实现思路看起来虽然简单，但是细节问题很多的，反而不容易写对。相反，递归实现就很简洁优美，下面就由浅入深，先从反转整个单链表说起。

一、递归反转整个链表

这也是力扣第 206 题「反转链表」，递归反转单链表的算法可能很多读者都听说过，这里详细介绍一下，直接看代码实现：

```
ListNode reverse(ListNode head) {
    if (head == null || head.next == null) {
        return head;
    }
    ListNode last = reverse(head.next);
    head.next.next = head;
    head.next = null;
```

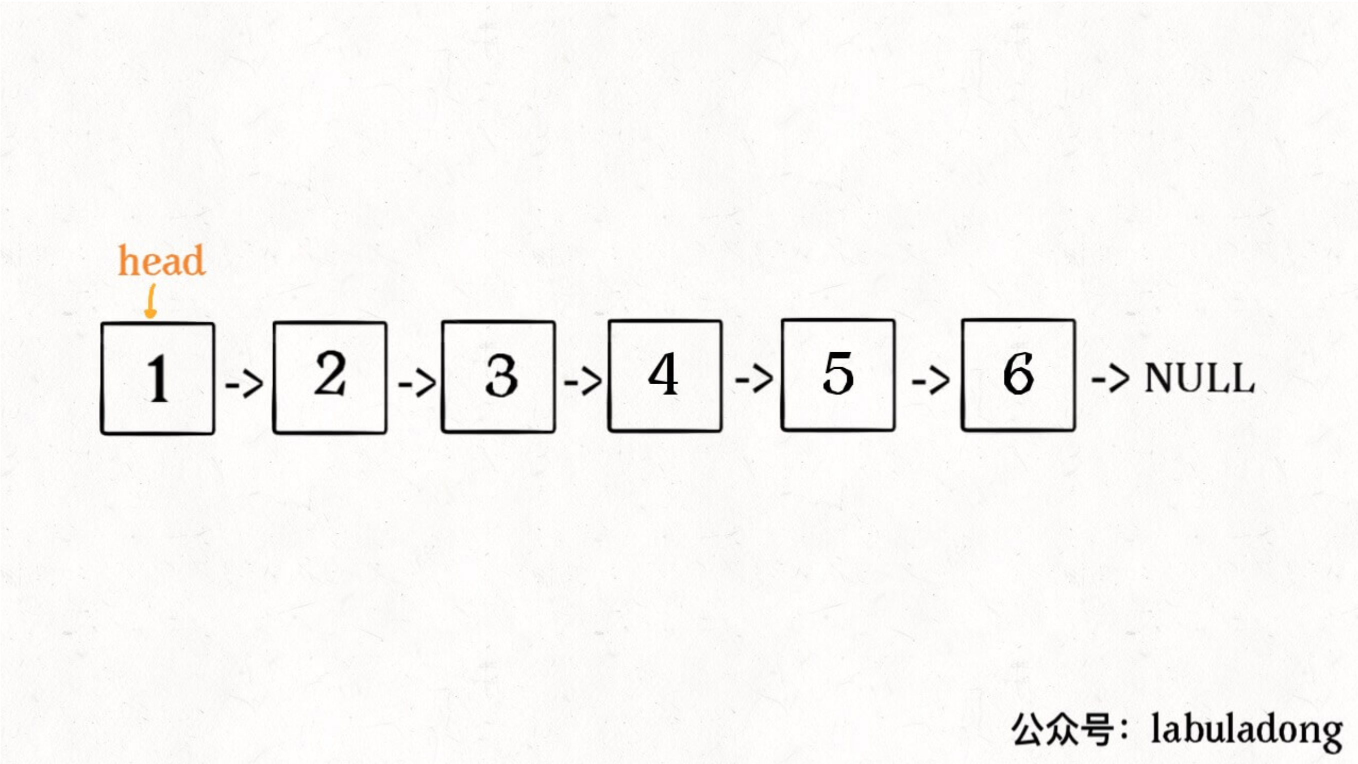
```
    return last;
}
```

看起来是不是感觉不知所云，完全不能理解这样为什么能够反转链表？这就对了，这个算法常常拿来显示递归的巧妙和优美，我们下面来详细解释一下这段代码。

对于递归算法，最重要的就是明确递归函数的定义。具体来说，我们的 `reverse` 函数定义是这样的：

输入一个节点 `head`，将「以 `head` 为起点」的链表反转，并返回反转之后的头结点。

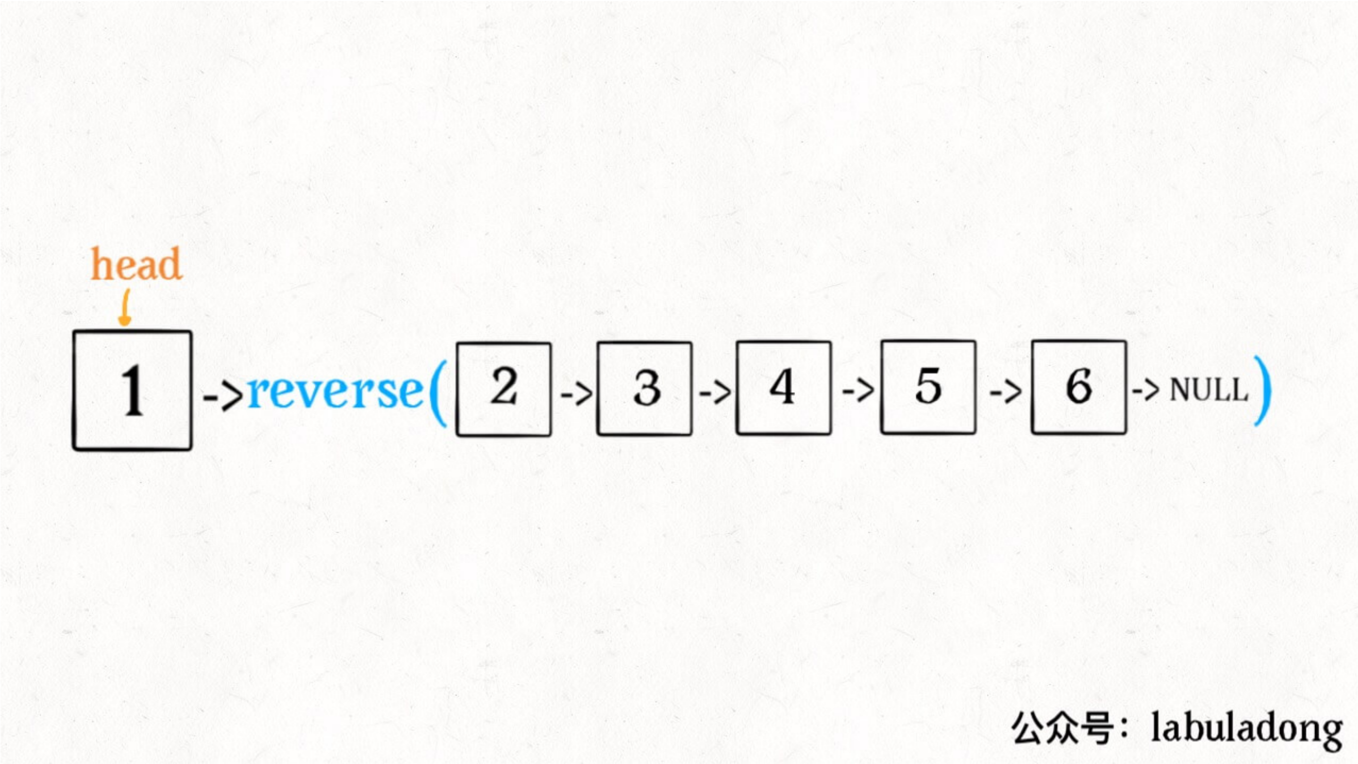
明白了函数的定义，再来看这个问题。比如说我们想反转这个链表：



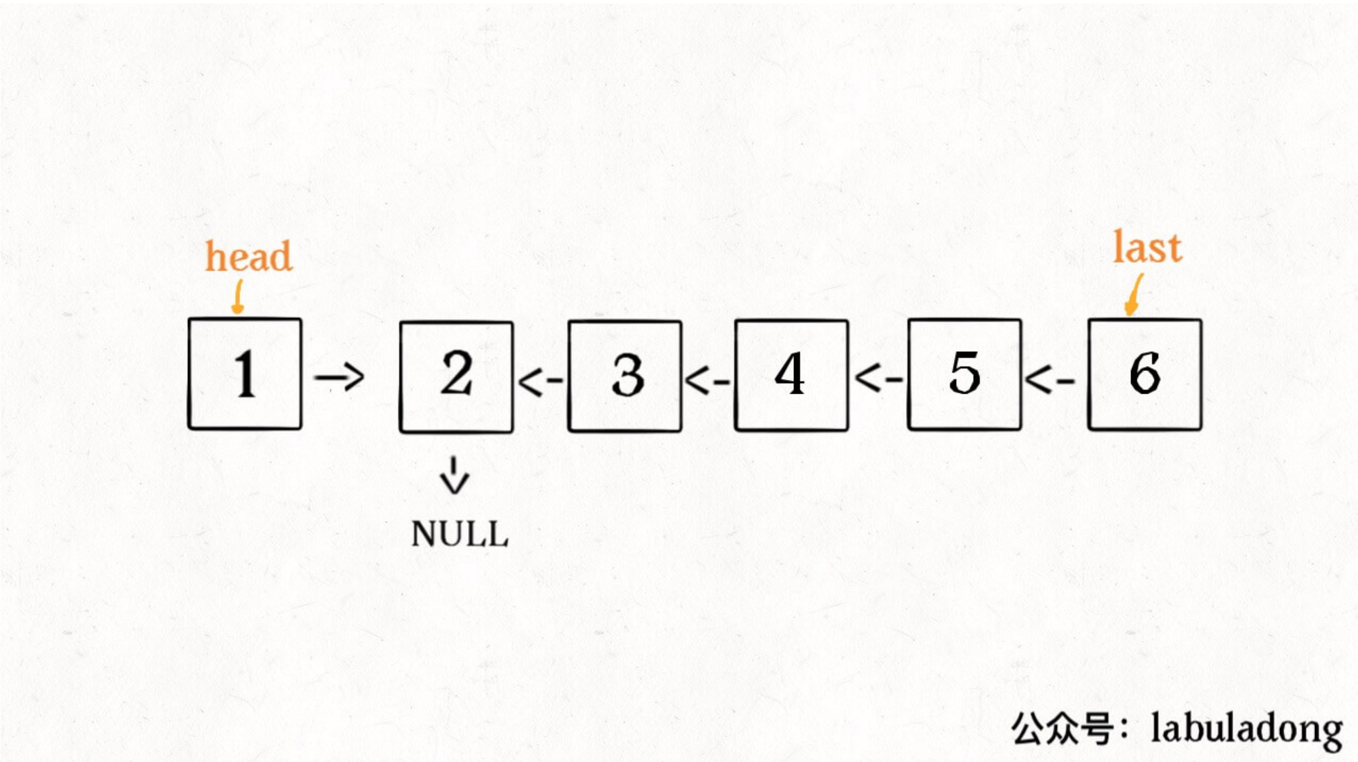
那么输入 `reverse(head)` 后，会在这里进行递归：

```
ListNode last = reverse(head.next);
```

不要跳进递归（你的脑袋能压几个栈呀？），而是要根据刚才的函数定义，来弄清楚这段代码会产生什么结果：



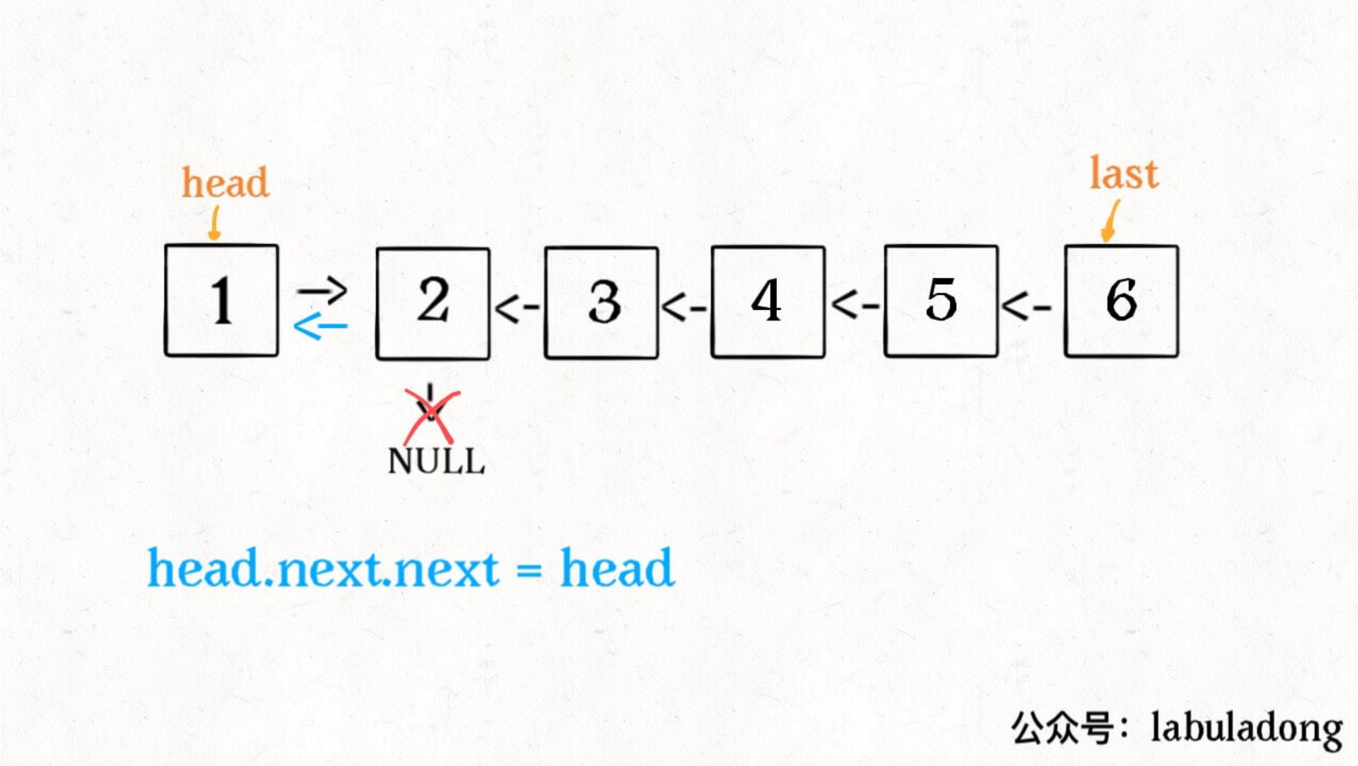
这个 `reverse(head.next)` 执行完成后，整个链表就成了这样：



并且根据函数定义，`reverse` 函数会返回反转之后的头结点，我们用变量 `last` 接收了。

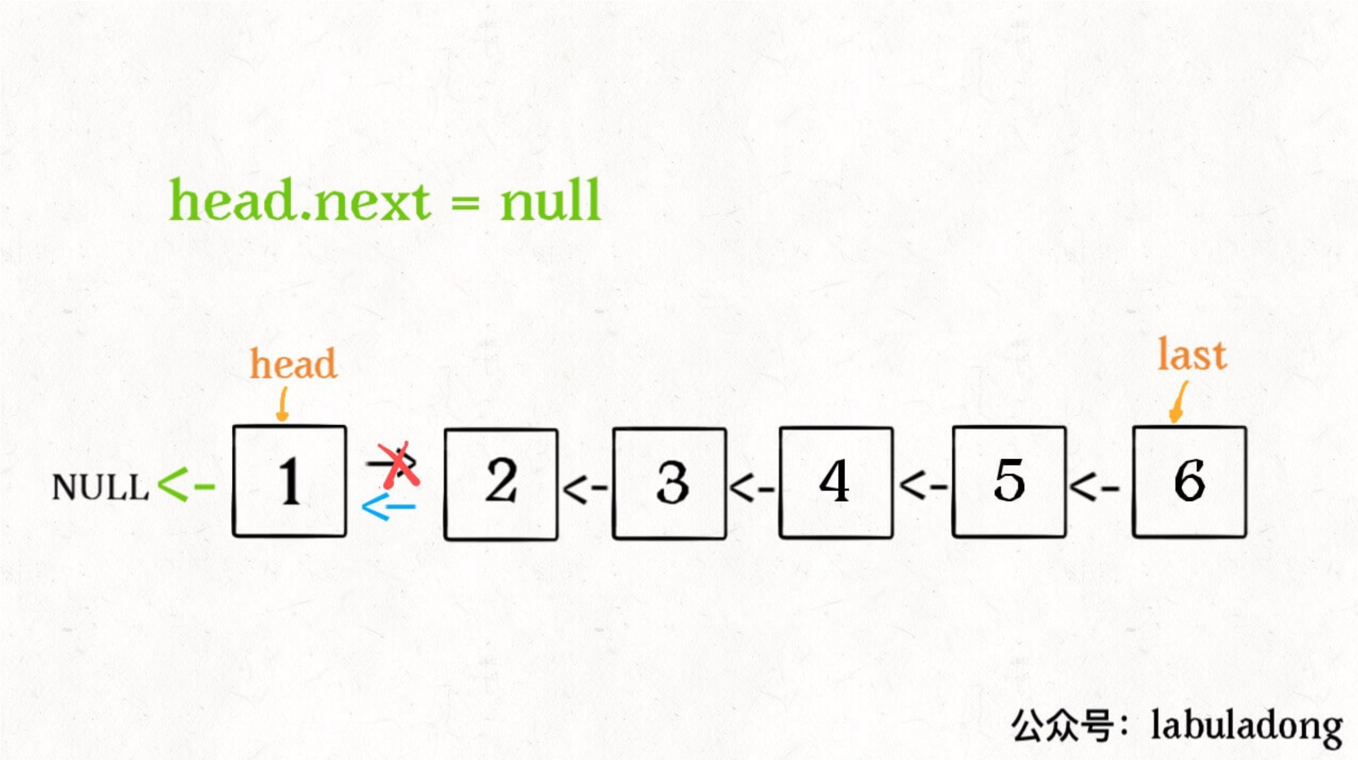
现在再来看下面的代码：

```
head.next.next = head;
```



接下来：

```
head.next = null;
return last;
```



神不神奇，这样整个链表就反转过来了！递归代码就是这么简洁优雅，不过其中有两个地方需要注意：

1、递归函数要有 base case，也就是这句：

```
if (head.next == null) return head;
```

意思是如果链表只有一个节点的时候反转也是它自己，直接返回即可。

2、当链表递归反转之后，新的头结点是 `last`，而之前的 `head` 变成了最后一个节点，别忘了链表的末尾要指向 null：

```
head.next = null;
```

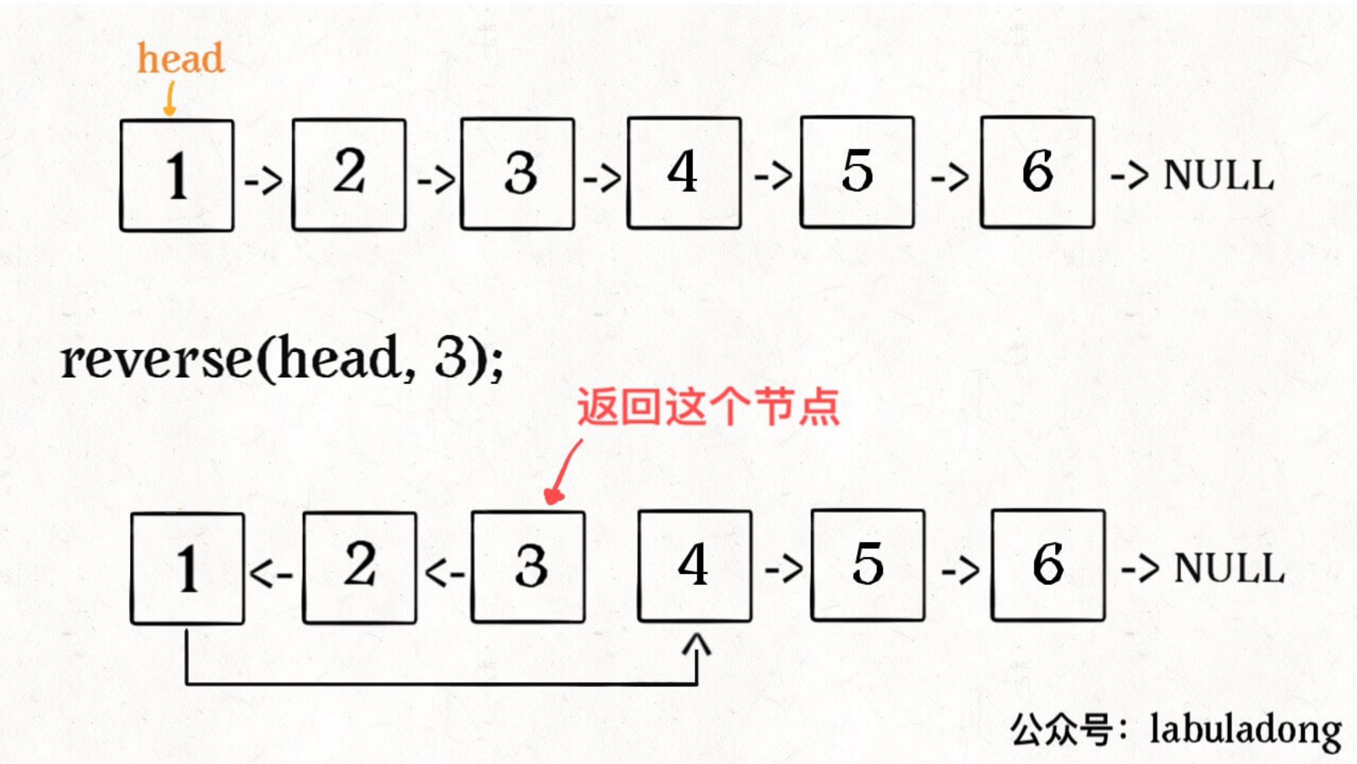
理解了这两点后，我们就可以进一步深入了，接下来的问题其实都是在这个算法上的扩展。

二、反转链表前 N 个节点

这次我们实现一个这样的函数：

```
// 将链表的前 n 个节点反转（n <= 链表长度）
ListNode reverseN(ListNode head, int n)
```

比如说对于下图链表，执行 `reverseN(head, 3)`：



解决思路和反转整个链表差不多，只要稍加修改即可：

```
ListNode successor = null; // 后驱节点

// 反转以 head 为起点的 n 个节点，返回新的头结点
```

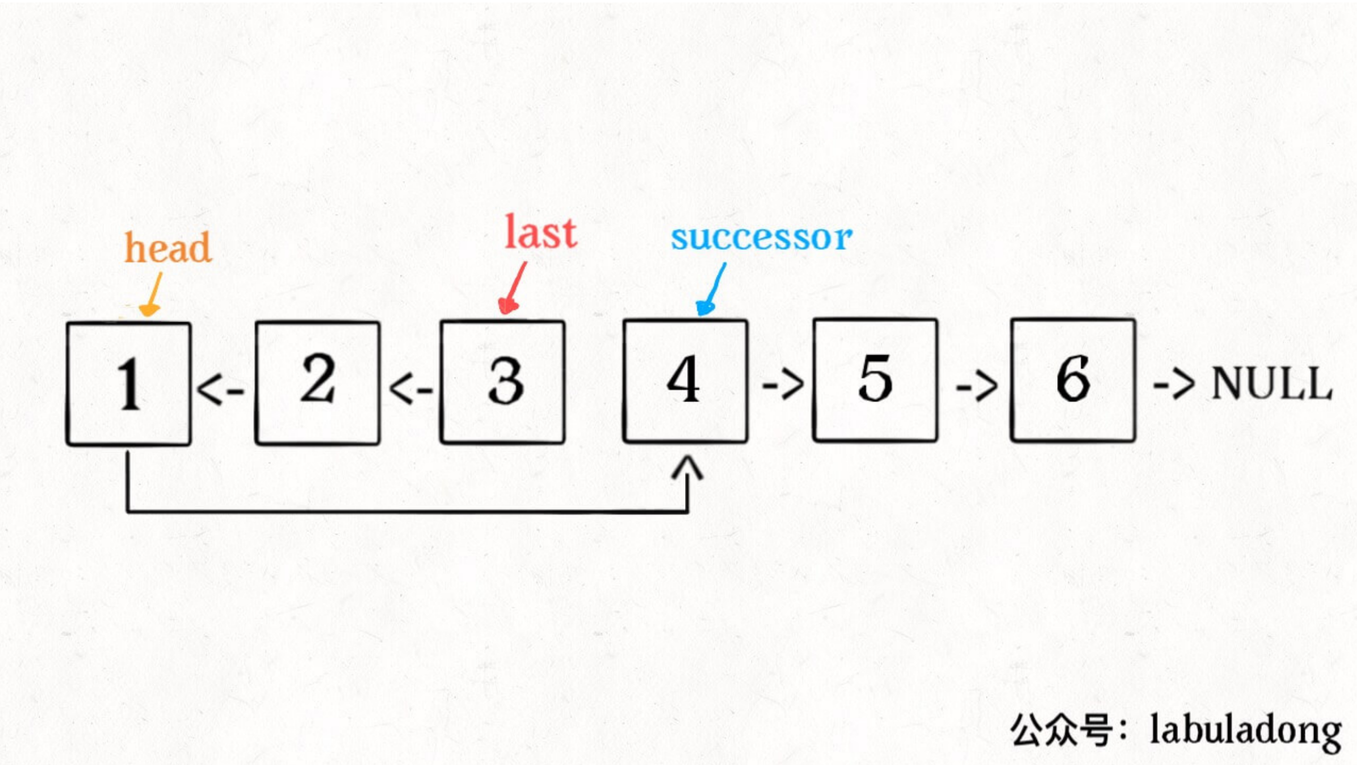
```java
ListNode reverseN(ListNode head, int n) {
    if (n == 1) {
        // 记录第 n + 1 个节点
        successor = head.next;
        return head;
    }
    // 以 head.next 为起点，需要反转前 n - 1 个节点
    ListNode last = reverseN(head.next, n - 1);

    head.next.next = head;
    // 让反转之后的 head 节点和后面的节点连起来
    head.next = successor;
    return last;
}
```

具体的区别：

1、base case 变为 `n == 1`，反转一个元素，就是它本身，同时**要记录后驱节点**。

2、刚才我们直接把 `head.next` 设置为 null，因为整个链表反转后原来的 `head` 变成了整个链表的最后一个节点。但现在 `head` 节点在递归反转之后不一定是最后一个节点了，所以要记录后驱 `successor`（第 n + 1 个节点），反转之后将 `head` 连接上。



OK，如果这个函数你也能看懂，就离实现「反转一部分链表」不远了。

三、反转链表的一部分

现在解决我们最开始提出的问题，给一个索引区间 `[m,n]`（索引从 1 开始），仅仅反转区间中的链表元素。

```java
ListNode reverseBetween(ListNode head, int m, int n)
```

首先，如果 `m == 1`，就相当于反转链表开头的 `n` 个元素嘛，也就是我们刚才实现的功能：

```
ListNode reverseBetween(ListNode head, int m, int n) {
    // base case
    if (m == 1) {
        // 相当于反转前 n 个元素
        return reverseN(head, n);
    }
    // ...
}
```

如果 `m != 1` 怎么办？如果我们把 `head` 的索引视为 1，那么我们是想从第 `m` 个元素开始反转对吧；如果把 `head.next` 的索引视为 1 呢？那么相对于 `head.next`，反转的区间应该是从第 `m - 1` 个元素开始的；那么对于 `head.next.next` 呢……

区别于迭代思想，这就是递归思想，所以我们可以完成代码：

```
ListNode reverseBetween(ListNode head, int m, int n) {
    // base case
    if (m == 1) {
        return reverseN(head, n);
    }
    // 前进到反转的起点触发 base case
    head.next = reverseBetween(head.next, m - 1, n - 1);
    return head;
}
```

至此，我们的最终大 BOSS 就被解决了。

四、最后总结

递归的思想相对迭代思想，稍微有点难以理解，处理的技巧是：不要跳进递归，而是利用明确的定义来实现算法逻辑。

处理看起来比较困难的问题，可以尝试化整为零，把一些简单的解法进行修改，解决困难的问题。

值得一提的是，递归操作链表并不高效。和迭代解法相比，虽然时间复杂度都是 O(N)，但是迭代解法的空间复杂度是 O(1)，而递归解法需要堆栈，空间复杂度是 O(N)。所以递归操作链表可以作为对递归算法的练习或者拿去和小伙伴装逼，但是考虑效率的话还是使用迭代算法更好。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

1.3 队列/栈

常言道：队列和栈是操作受限的数据结构。

为什么这样说呢？因为队列和栈底层就是数组和链表封装的。

数组和链表本身的操作可以花里胡哨，但队列和栈只给你暴露头尾操作的 API，这可不就是操作受限的数据结构么。

就算法题的角度来看，队列和栈的题目并不是很多，队列主要用在 BFS 算法，栈主要用在括号相关的问题。

公众号标签：队列和栈

队列实现栈以及栈实现队列

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

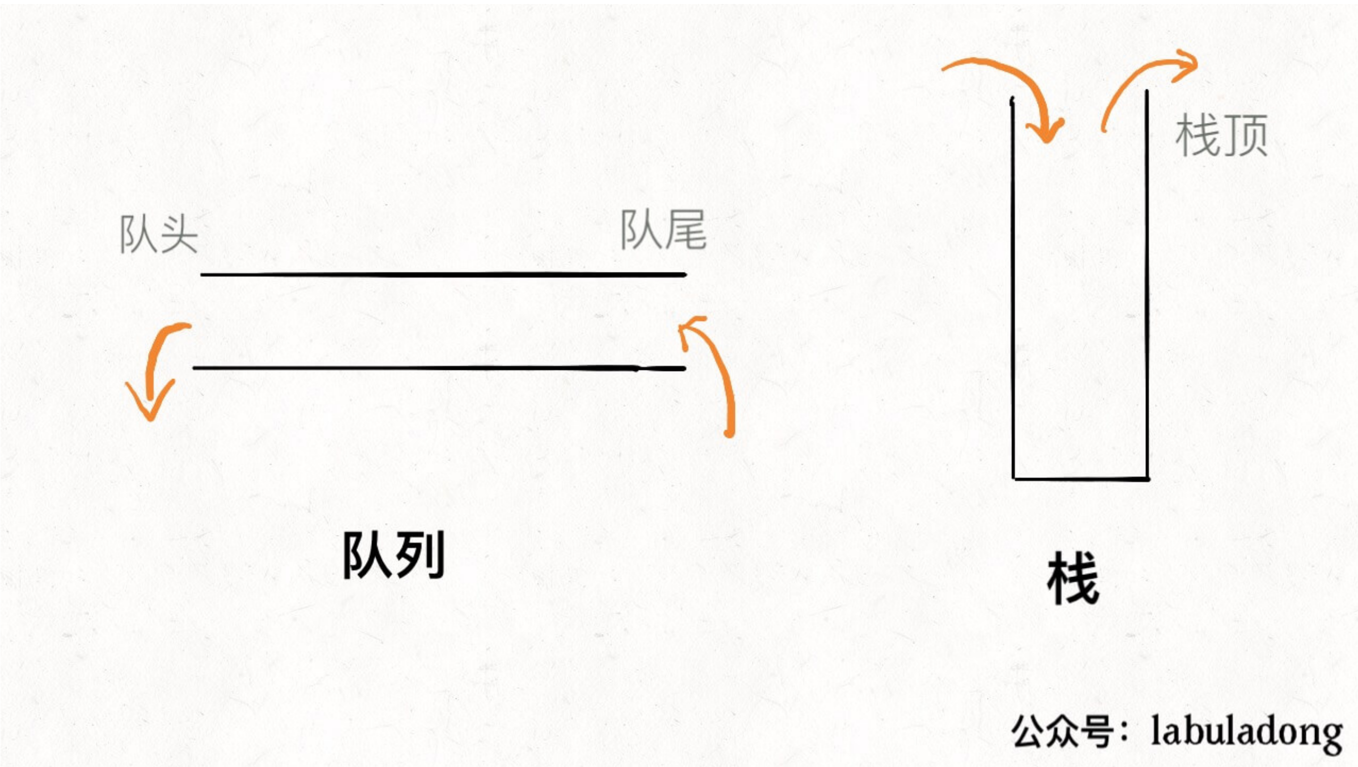
学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

232. 用栈实现队列（简单）

225. 用队列实现栈（简单）

队列是一种先进先出的数据结构，栈是一种先进后出的数据结构，形象一点就是这样：



这两种数据结构底层其实都是数组或者链表实现的，只是 API 限定了它们的特性，那么今天就来看看如何使用「栈」的特性来实现一个「队列」，如何用「队列」实现一个「栈」。

一、用栈实现队列

力扣第 232 题「用栈实现队列」让我们实现的 API 如下：

```java
class MyQueue {

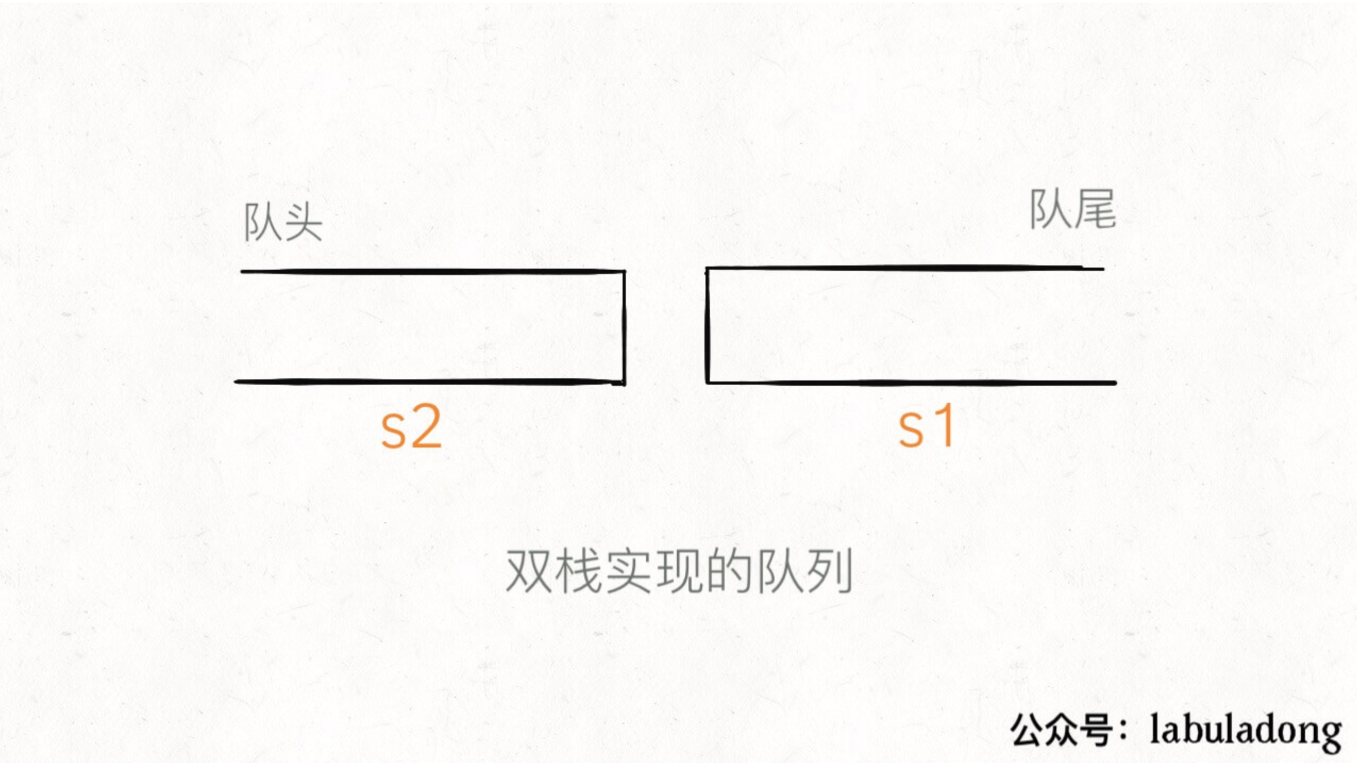
    /** 添加元素到队尾 */
    public void push(int x);

    /** 删除队头的元素并返回 */
    public int pop();

    /** 返回队头元素 */
    public int peek();

    /** 判断队列是否为空 */
    public boolean empty();
}
```

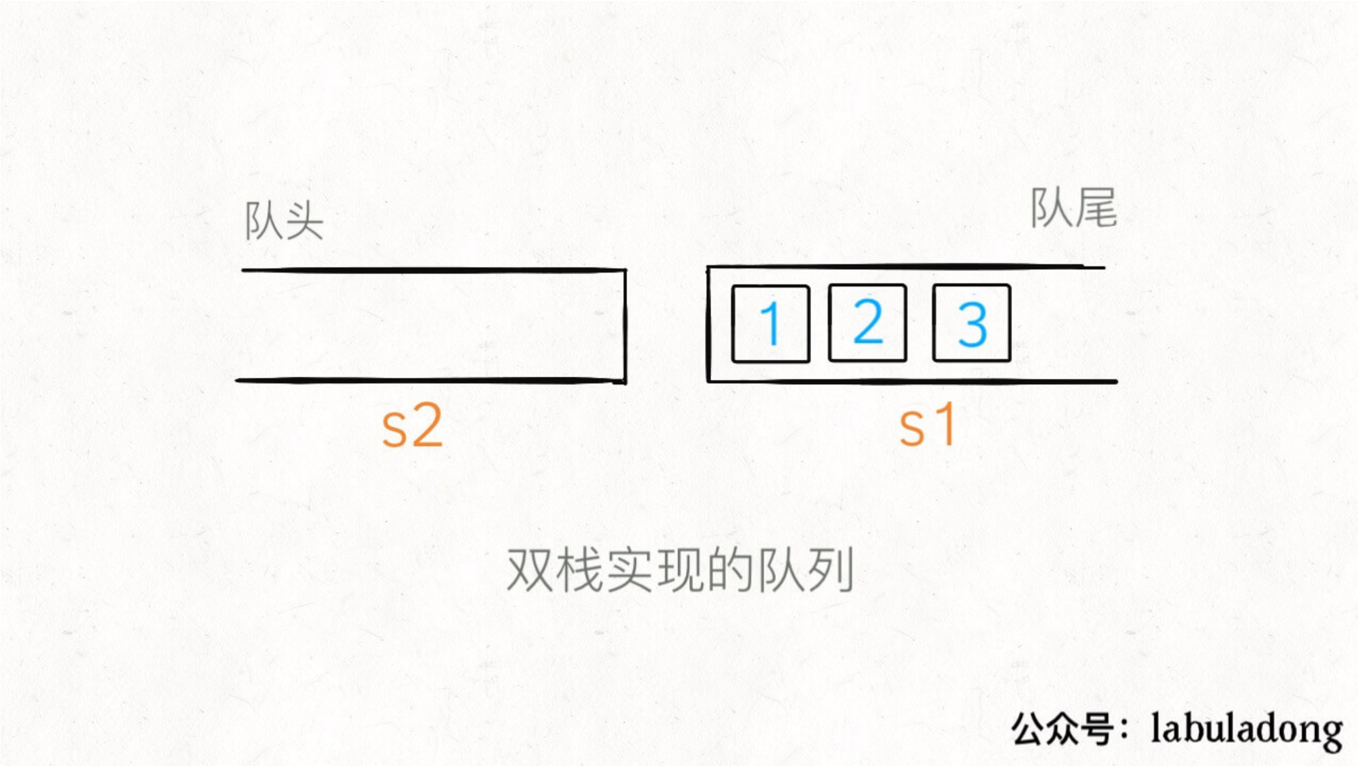
我们使用两个栈 `s1, s2` 就能实现一个队列的功能（这样放置栈可能更容易理解）：




```java
class MyQueue {
    private Stack<Integer> s1, s2;

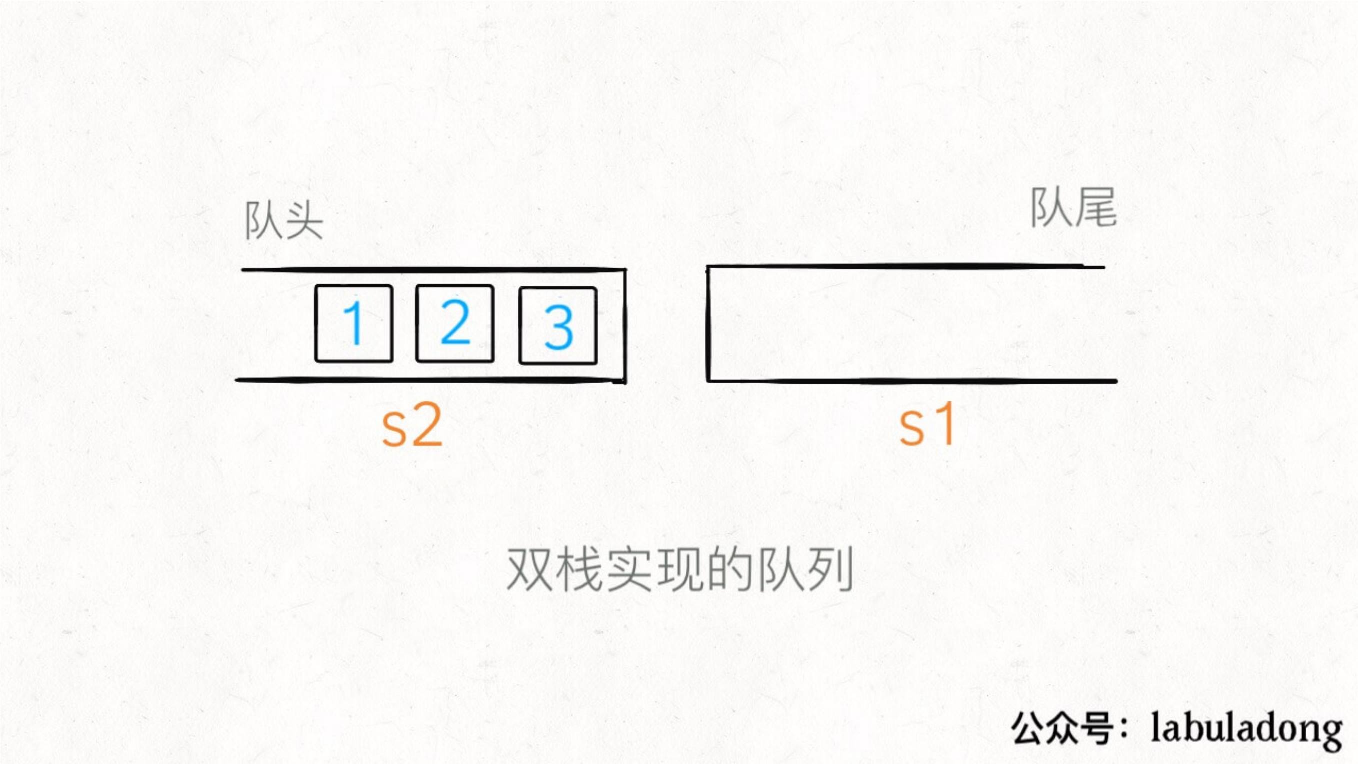
    public MyQueue() {
        s1 = new Stack<>();
        s2 = new Stack<>();
    }
    // ...
}
```

当调用 push 让元素入队时，只要把元素压入 s1 即可，比如说 push 进 3 个元素分别是 1,2,3，那么底层结构就是这样：




```java
/** 添加元素到队尾 */
public void push(int x) {
    s1.push(x);
}
```

那么如果这时候使用 peek 查看队头的元素怎么办呢？按道理队头元素应该是 1，但是在 s1 中 1 被压在栈底，现在就要轮到 s2 起到一个中转的作用了：当 s2 为空时，可以把 s1 的所有元素取出再添加进 s2，**这时候 s2 中元素就是先进先出顺序了**。




```java
/** 返回队头元素 */
public int peek() {
    if (s2.isEmpty())
        // 把 s1 元素压入 s2
        while (!s1.isEmpty())
            s2.push(s1.pop());
    return s2.peek();
}
```

同理，对于 `pop` 操作，只要操作 `s2` 就可以了。

```java
/** 删除队头的元素并返回 */
public int pop() {
    // 先调用 peek 保证 s2 非空
    peek();
    return s2.pop();
}
```

最后，如何判断队列是否为空呢？如果两个栈都为空的话，就说明队列为空：

```java
/** 判断队列是否为空 */
public boolean empty() {
    return s1.isEmpty() && s2.isEmpty();
}
```

至此，就用栈结构实现了一个队列，核心思想是利用两个栈互相配合。

值得一提的是，这几个操作的时间复杂度是多少呢？有点意思的是 peek 操作，调用它时可能触发 while 循环，这样的话时间复杂度是 O(N)，但是大部分情况下 while 循环不会被触发，时间复杂度是 O(1)。由于 pop 操作调用了 peek，它的时间复杂度和 peek 相同。

像这种情况，可以说它们的**最坏时间复杂度**是 O(N)，因为包含 while 循环，**可能**需要从 s1 往 s2 搬移元素。

但是它们的**均摊时间复杂度**是 O(1)，这个要这么理解：对于一个元素，最多只可能被搬运一次，也就是说 peek 操作平均到每个元素的时间复杂度是 O(1)。

二、用队列实现栈

如果说双栈实现队列比较巧妙，那么用队列实现栈就比较简单粗暴了，只需要一个队列作为底层数据结构。

力扣第 25 题「用队列实现栈」让我们实现如下 API：

```
class MyStack {

    /** 添加元素到栈顶 */
    public void push(int x);

    /** 删除栈顶的元素并返回 */
    public int pop();

    /** 返回栈顶元素 */
    public int top();

    /** 判断栈是否为空 */
    public boolean empty();
}
```

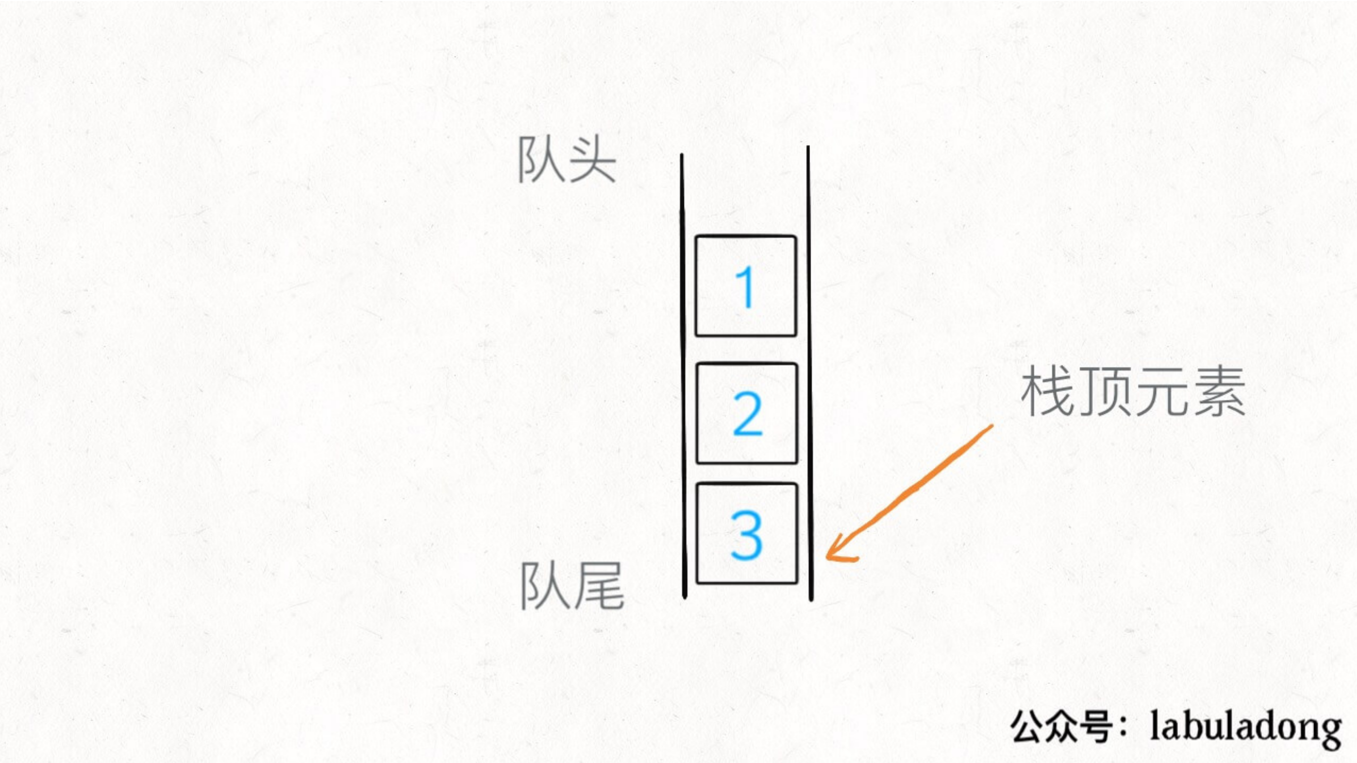
先说 push API，直接将元素加入队列，同时记录队尾元素，因为队尾元素相当于栈顶元素，如果要 top 查看栈顶元素的话可以直接返回：

```
class MyStack {
    Queue<Integer> q = new LinkedList<>();
    int top_elem = 0;

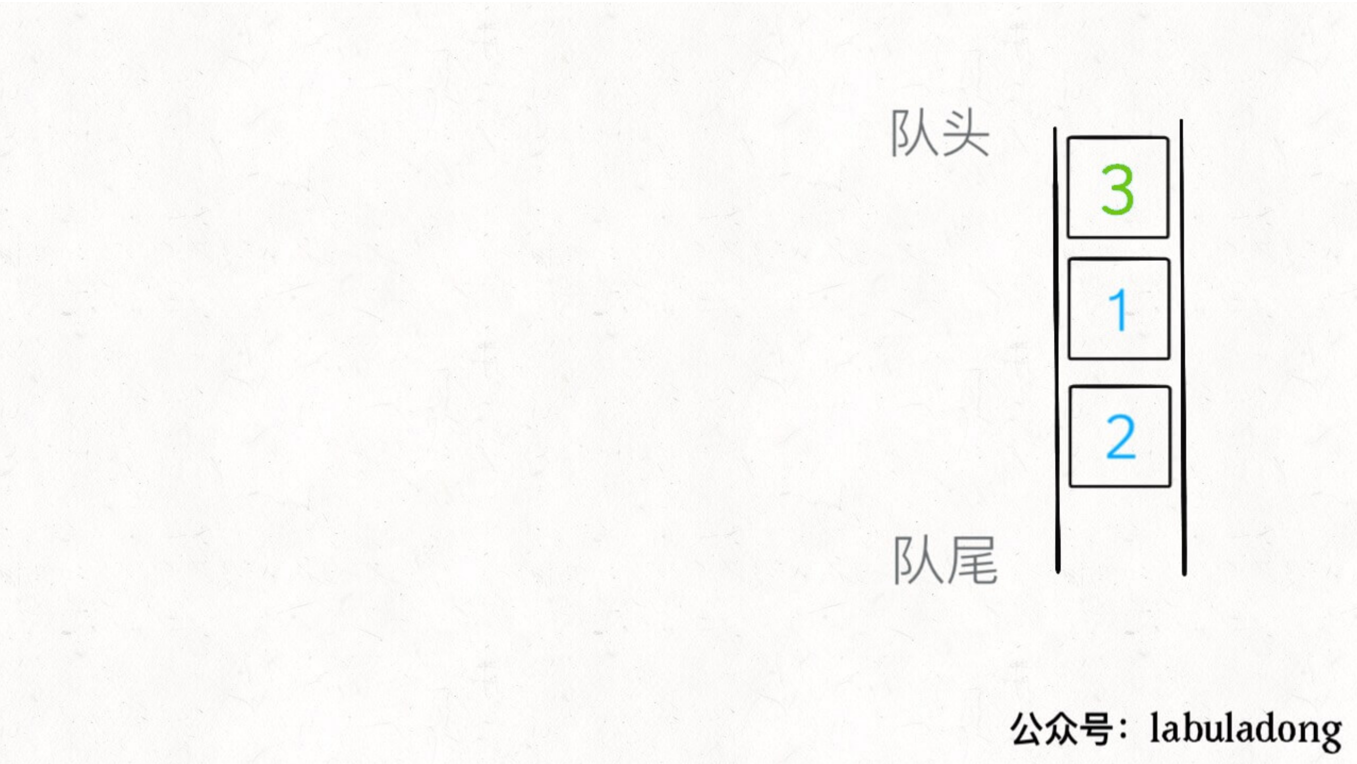
    /** 添加元素到栈顶 */
    public void push(int x) {
        // x 是队列的队尾，是栈的栈顶
        q.offer(x);
        top_elem = x;
    }

    /** 返回栈顶元素 */
    public int top() {
        return top_elem;
    }
}
```

我们的底层数据结构是先进先出的队列，每次 pop 只能从队头取元素；但是栈是后进先出，也就是说 pop API 要从队尾取元素：



解决方法简单粗暴，把队列前面的都取出来再加入队尾，让之前的队尾元素排到队头，这样就可以取出了：




```
/** 删除栈顶的元素并返回 */
public int pop() {
    int size = q.size();
```

```
        while (size > 1) {
            q.offer(q.poll());
            size--;
        }
        // 之前的队尾元素已经到了队头
        return q.poll();
    }
```

这样实现还有一点小问题就是，原来的队尾元素被提到队头并删除了，但是 `top_elem` 变量没有更新，我们还需要一点小修改：

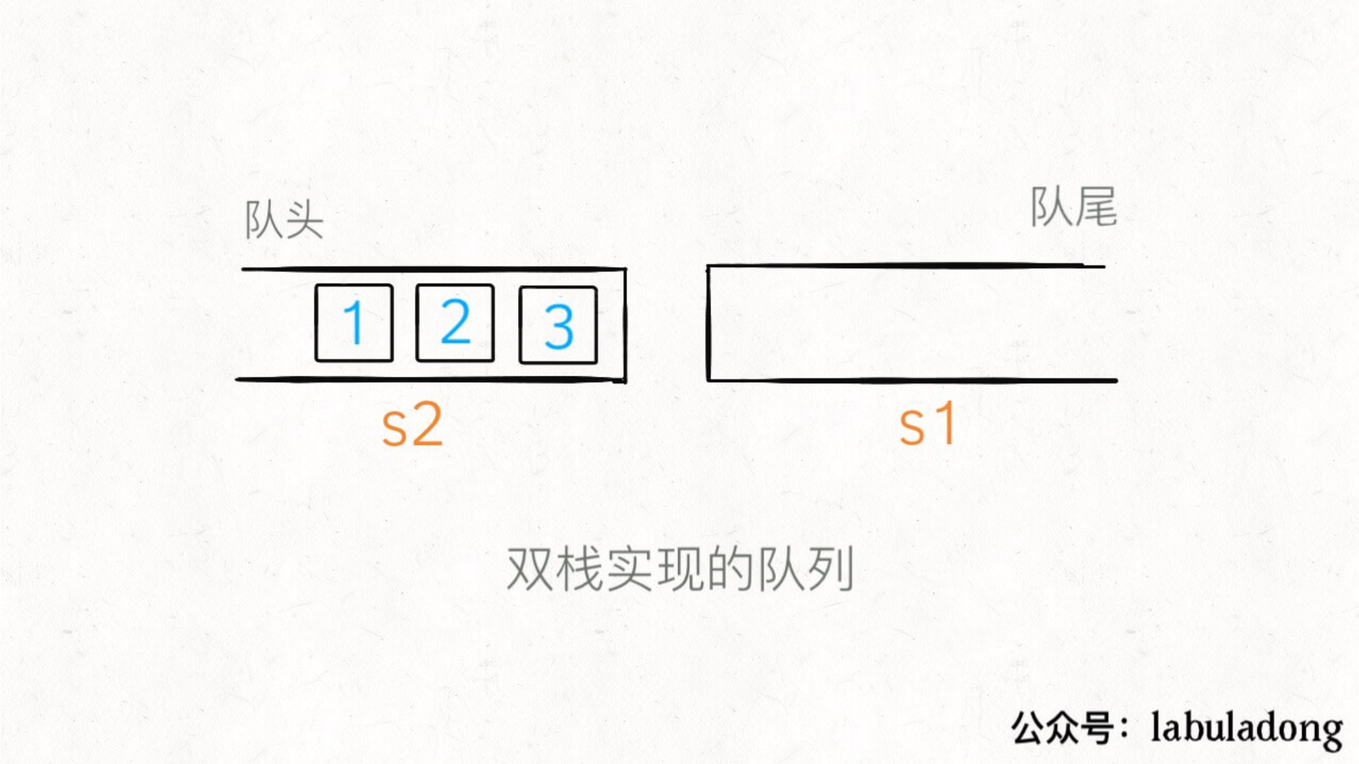
```
/** 删除栈顶的元素并返回 */
public int pop() {
    int size = q.size();
    // 留下队尾 2 个元素
    while (size > 2) {
        q.offer(q.poll());
        size--;
    }
    // 记录新的队尾元素
    top_elem = q.peek();
    q.offer(q.poll());
    // 删除之前的队尾元素
    return q.poll();
}
```

最后，API `empty` 就很容易实现了，只要看底层的队列是否为空即可：

```
/** 判断栈是否为空 */
public boolean empty() {
    return q.isEmpty();
}
```

很明显，用队列实现栈的话，`pop` 操作时间复杂度是 O(N)，其他操作都是 O(1)。

个人认为，用队列实现栈是没啥亮点的问题，但是**用双栈实现队列是值得学习的**。



从栈 s1 搬运元素到 s2 之后，元素在 s2 中就变成了队列的先进先出顺序，这个特性有点类似「负负得正」，确实不太容易想到。

希望本文对你有帮助。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



一文秒杀三道括号题目

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

20. 有效的括号（简单）

921. 使括号有效的最小添加（中等）

1541. 平衡括号串的最少插入（中等）

判断有效括号串

对括号的有效性判断多次在笔试中出现，现实中也很常见，比如说我们写的代码，编辑器会检查括号是否正确闭合。而且我们的代码可能会包含三种括号 `[](){}`，判断起来有一点难度。

来看一看力扣第 20 题「有效的括号」，输入一个字符串，其中包含 `[](){}` 六种括号，请你判断这个字符串组成的括号是否有效。

举几个例子：

```
Input: "()[]{}"
Output: true

Input: "([)]"
Output: false

Input: "{[]}"
Output: true
```

解决这个问题之前，我们先降低难度，思考一下，**如果只有一种括号 `()`**，应该如何判断字符串组成的括号是否有效呢？

假设字符串中只有圆括号，如果想让括号字符串有效，那么必须做到：

每个右括号 `)` 的左边必须有一个左括号 `(` 和它匹配。

比如说字符串 `()))((` 中，中间的两个右括号**左边**就没有左括号匹配，所以这个括号组合是无效的。

那么根据这个思路，我们可以写出算法：

```
bool isValid(string str) {
    // 待匹配的左括号数量
    int left = 0;
    for (int i = 0; i < str.size(); i++) {
        if (s[i] == '(') {
            left++;
        } else {
            // 遇到右括号
            left--;
        }

        // 右括号太多
        if (left == -1)
            return false;
    }
    // 是否所有的左括号都被匹配了
    return left == 0;
}
```

如果只有圆括号，这样就能正确判断有效性。对于三种括号的情况，我一开始想模仿这个思路，定义三个变量 `left1`，`left2`，`left3` 分别处理每种括号，虽然要多写不少 if else 分支，但是似乎可以解决问题。

但实际上直接照搬这种思路是不行的，比如说只有一个括号的情况下 `(())` 是有效的，但是多种括号的情况下，`[(])` 显然是无效的。

仅仅记录每种左括号出现的次数已经不能做出正确判断了，我们要加大存储的信息量，可以利用栈来模仿类似的思路。栈是一种先进后出的数据结构，处理括号问题的时候尤其有用。

我们这道题就用一个名为 `left` 的栈代替之前思路中的 `left` 变量，**遇到左括号就入栈，遇到右括号就去栈中寻找最近的左括号，看是否匹配**：

```
bool isValid(string str) {
    stack<char> left;
    for (char c : str) {
        if (c == '(' || c == '{' || c == '[')
            left.push(c);
        else { // 字符 c 是右括号
            if (!left.empty() && leftOf(c) == left.top())
                left.pop();
            else
                // 和最近的左括号不匹配
                return false;
        }
    }
    // 是否所有的左括号都被匹配了
    return left.empty();
}
```

```
char leftOf(char c) {
    if (c == '}') return '{';
    if (c == ')') return '(';
    return '[';
}
```

接下来讲另外两个常见的问题，如何通过最小的插入次数将括号变成有效的？

平衡括号串（一）

先来个简单的，力扣第 921 题「使括号有效的最少添加」：

给你输入一个字符串 `s`，你可以在其中的任意位置插入左括号 `(` 或者右括号 `)`，请问你最少需要几次插入才能使得 `s` 变成一个有效的括号串？

比如说输入 `s = "())("`，算法应该返回 2，因为我们至少需要插入两次把 `s` 变成 `"(())()"`，这样每个左括号都有一个右括号匹配，`s` 是一个有效的括号串。

这其实和前文的判断括号有效性非常类似，我们直接看代码：

```
int minAddToMakeValid(string s) {
    // res 记录插入次数
    int res = 0;
    // need 变量记录右括号的需求量
    int need = 0;

    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '(') {
            // 对右括号的需求 + 1
            need++;
        }

        if (s[i] == ')') {
            // 对右括号的需求 - 1
            need--;

            if (need == -1) {
                need = 0;
                // 需插入一个左括号
                res++;
            }
        }
    }

    return res + need;
}
```

这段代码就是最终解法，**核心思路是以左括号为基准，通过维护对右括号的需求数 `need`，来计算最小的插入次数**。需要注意两个地方：

1、当 `need == -1` 的时候意味着什么？

因为只有遇到右括号 `)` 的时候才会 `need--`，`need == -1` 意味着右括号太多了，所以需要插入左括号。

比如说 `s = "))"` 这种情况，需要插入 2 个左括号，使得 `s` 变成 `"()()"`，才是一个有效括号串。

2、算法为什么返回 `res + need`？

因为 `res` 记录的左括号的插入次数，`need` 记录了右括号的需求，当 for 循环结束后，若 `need` 不为 0，那么就意味着右括号还不够，需要插入。

比如说 `s = "))("` 这种情况，插入 2 个左括号之后，还要再插入 1 个右括号，使得 `s` 变成 `"()()()"`，才是一个有效括号串。

以上就是这道题的思路，接下来我们看一道进阶题目，如果左右括号不是 1:1 配对，会出现什么问题呢？

平衡括号串（二）

这是力扣第 1541 题「平衡括号字符串的最少插入次数」：

现在假设 1 个左括号需要匹配 2 个右括号才叫做有效的括号组合，那么给你输入一个括号串 `s`，请问你如何计算使得 `s` 有效的最小插入次数呢？

核心思路还是和刚才一样，通过一个 `need` 变量记录对右括号的需求数，根据 `need` 的变化来判断是否需要插入。

第一步，我们按照刚才的思路正确维护 `need` 变量：

```
int minInsertions(string s) {
    // need 记录需右括号的需求量
    int res = 0, need = 0;

    for (int i = 0; i < s.size(); i++) {
        // 一个左括号对应两个右括号
        if (s[i] == '(') {
            need += 2;
        }

        if (s[i] == ')') {
            need--;
        }
    }

    return res + need;
}
```

现在想一想，当 `need` 为什么值的时候，我们可以确定需要进行插入？

首先，类似第一题，当 `need == -1` 时，意味着我们遇到一个多余的右括号，显然需要插入一个左括号。

比如说当 `s = ")"`，我们肯定需要插入一个左括号让 `s = "()"`，但是由于一个左括号需要两个右括号，所以对右括号的需求量变为 1：

```
if (s[i] == ')') {
    need--;
    // 说明右括号太多了
    if (need == -1) {
        // 需要插入一个左括号
        res++;
        // 同时，对右括号的需求变为 1
        need = 1;
    }
}
```

另外，当遇到左括号时，若对右括号的需求量为奇数，需要插入 1 个右括号。因为一个左括号需要两个右括号嘛，右括号的需求必须是偶数，这一点也是本题的难点。

所以遇到左括号时要做如下判断：

```
if (s[i] == '(') {
    need += 2;
    if (need % 2 == 1) {
        // 插入一个右括号
        res++;
        // 对右括号的需求减一
        need--;
    }
}
```

综上，我们可以写出正确的代码：

```
int minInsertions(string s) {
    int res = 0, need = 0;

    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '(') {
            need += 2;
            if (need % 2 == 1) {
                res++;
                need--;
            }
        }

        if (s[i] == ')') {
            need--;
            if (need == -1) {
                res++;
                need = 1;
            }
        }
    }
```

```
        return res + need;
    }
```

综上，三道括号相关的问题就解决了，其实我们前文 有效括号生成算法 也是括号相关的问题，但是使用的回溯算法技巧，和本文的几道题差别还是蛮大的，有兴趣的读者可以去看看。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



单调栈结构解决三道算法题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

496. 下一个更大元素I（简单）

503. 下一个更大元素II（中等）

739. 每日温度（中等）

栈（stack）是很简单的一种数据结构，先进后出的逻辑顺序，符合某些问题的特点，比如说函数调用栈。

单调栈实际上就是栈，只是利用了一些巧妙的逻辑，使得每次新元素入栈后，栈内的元素都保持有序（单调递增或单调递减）。

听起来有点像堆（heap）？不是的，单调栈用途不太广泛，只处理一种典型的问题，叫做 Next Greater Element。本文用讲解单调队列的算法模版解决这类问题，并且探讨处理「循环数组」的策略。

单调栈模板

现在给你出这么一道题：

给你一个数组 `nums`，请你返回一个等长的结果数组，结果数组中对应索引存储着下一个更大元素，如果没有更大的元素，就存 -1。

函数签名如下：

```
vector<int> nextGreaterElement(vector<int>& nums);
```

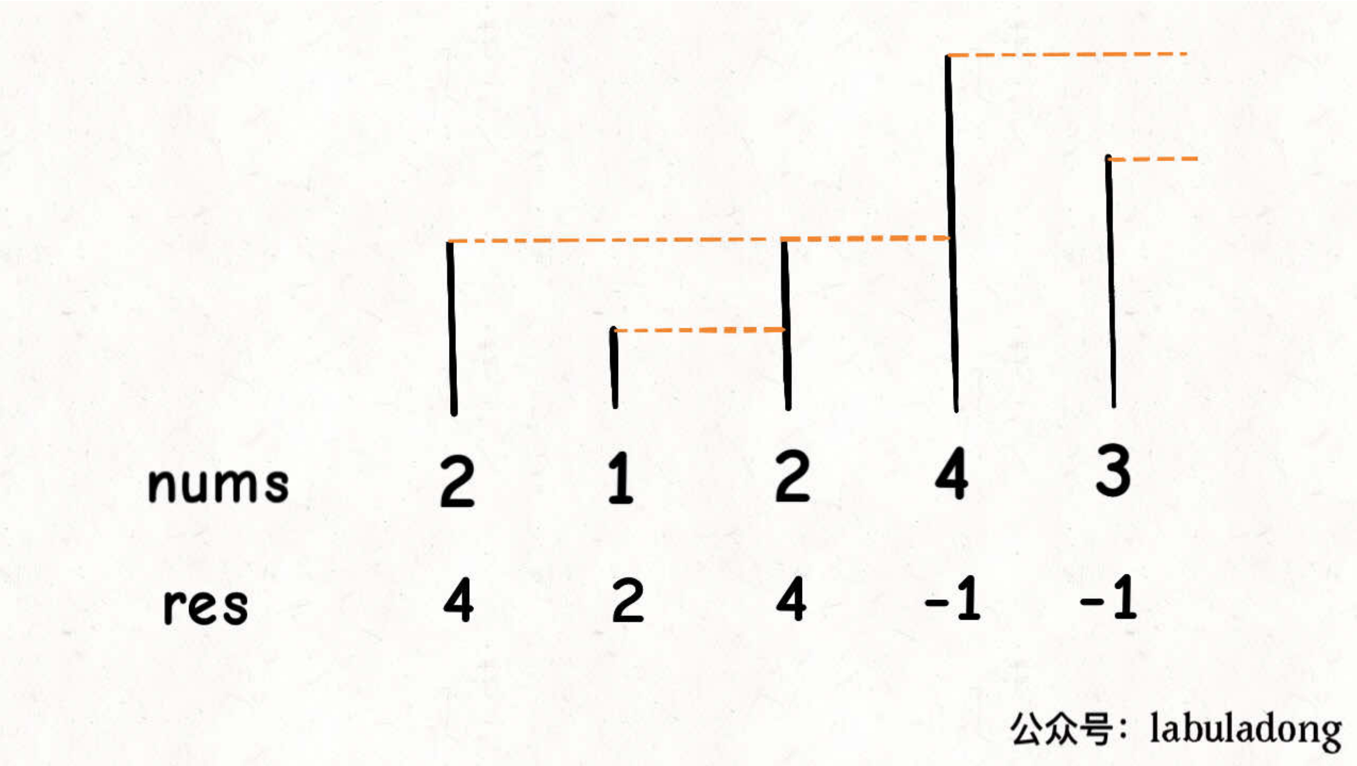
比如说，输入一个数组 `nums = [2,1,2,4,3]`，你返回数组 `[4,2,4,-1,-1]`。

解释：第一个 2 后面比 2 大的数是 4; 1 后面比 1 大的数是 2；第二个 2 后面比 2 大的数是 4; 4 后面没有比 4 大的数，填 -1；3 后面没有比 3 大的数，填 -1。

这道题的暴力解法很好想到，就是对每个元素后面都进行扫描，找到第一个更大的元素就行了。但是暴力解法的时间复杂度是 `O(n^2)`。

这个问题可以这样抽象思考：把数组的元素想象成并列站立的人，元素大小想象成人的身高。这些人面对你站成一列，如何求元素「2」的 Next Greater Number 呢？

很简单，如果能够看到元素「2」，那么他后面可见的第一个人就是「2」的 Next Greater Number，因为比「2」小的元素身高不够，都被「2」挡住了，第一个露出来的就是答案。



这个情景很好理解吧？带着这个抽象的情景，先来看下代码。

```
vector<int> nextGreaterElement(vector<int>& nums) {
    vector<int> res(nums.size()); // 存放答案的数组
    stack<int> s;
    // 倒着往栈里放
    for (int i = nums.size() - 1; i >= 0; i--) {
        // 判定个子高矮
        while (!s.empty() && s.top() <= nums[i]) {
            // 矮个起开，反正也被挡着了。。。
            s.pop();
        }
        // nums[i] 身后的 next great number
        res[i] = s.empty() ? -1 : s.top();
        s.push(nums[i]);
    }
    return res;
}
```

这就是单调队列解决问题的模板。for 循环要从后往前扫描元素，因为我们借助的是栈的结构，倒着入栈，其实是正着出栈。while 循环是把两个「个子高」元素之间的元素排除，因为他们的存在没有意义，前面挡着个「更高」的元素，所以他们不可能被作为后续进来的元素的 Next Great Number 了。

这个算法的时间复杂度不是那么直观，如果你看到 for 循环嵌套 while 循环，可能认为这个算法的复杂度也是 `O(n^2)`，但是实际上这个算法的复杂度只有 `O(n)`。

分析它的时间复杂度，要从整体来看：总共有 `n` 个元素，每个元素都被 `push` 入栈了一次，而最多会被 `pop` 一次，没有任何冗余操作。所以总的计算规模是和元素规模 `n` 成正比的，也就是 `O(n)` 的复杂度。

问题变形

单调栈的使用技巧差不多了，来一个简单的变形，力扣第 739 题「每日温度」：

给你一个数组 `T`，这个数组存放的是近几天的天气气温，你返回一个等长的数组，计算：**对于每一天，你还要至少等多少天才能等到一个更暖和的气温；如果等不到那一天，填 0**。

函数签名如下：

```
vector<int> dailyTemperatures(vector<int>& T);
```

比如说给你输入 `T = [73,74,75,71,69,76]`，你返回 `[1,1,3,2,1,0]`。

解释：第一天 73 华氏度，第二天 74 华氏度，比 73 大，所以对于第一天，只要等一天就能等到一个更暖和的气温，后面的同理。

这个问题本质上也是找 Next Greater Number，只不过现在不是问你 Next Greater Number 是多少，而是问你当前距离 Next Greater Number 的距离而已。

相同的思路，直接调用单调栈的算法模板，稍作改动就可以，直接上代码吧：

```
vector<int> dailyTemperatures(vector<int>& T) {
    vector<int> res(T.size());
    // 这里放元素索引，而不是元素
    stack<int> s;
    /* 单调栈模板 */
    for (int i = T.size() - 1; i >= 0; i--) {
        while (!s.empty() && T[s.top()] <= T[i]) {
            s.pop();
        }
        // 得到索引间距
        res[i] = s.empty() ? 0 : (s.top() - i);
        // 将索引入栈，而不是元素
        s.push(i);
    }
    return res;
}
```

单调栈讲解完毕，下面开始另一个重点：如何处理「循环数组」。

如何处理环形数组

同样是 Next Greater Number，现在假设给你的数组是个环形的，如何处理？力扣第 503 题「下一个更大元素 II」就是这个问题：

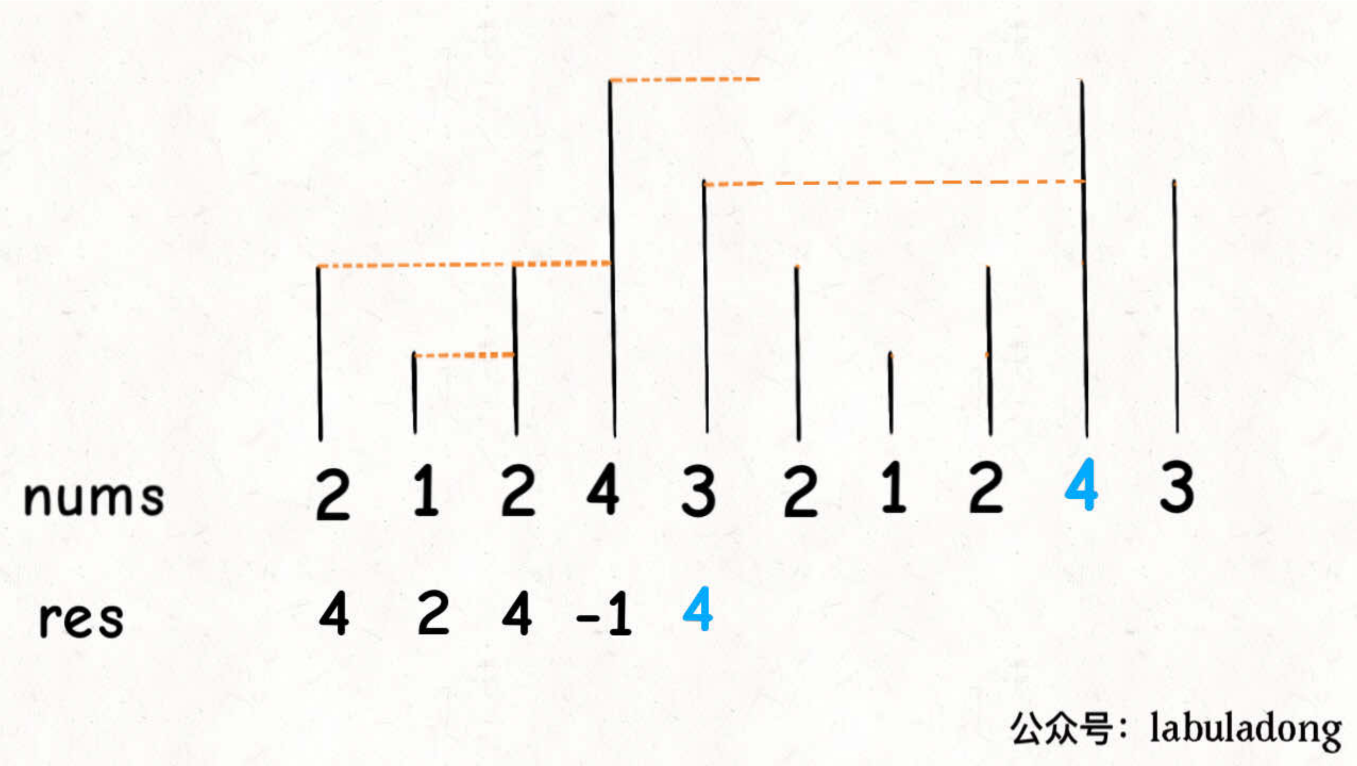
比如输入一个数组 [2,1,2,4,3]，你返回数组 [4,2,4,-1,4]。拥有了环形属性，**最后一个元素 3 绕了一圈后找到了比自己大的元素 4**。

一般是通过 % 运算符求模（余数），来获得环形特效：

```
int[] arr = {1,2,3,4,5};
int n = arr.length, index = 0;
while (true) {
    print(arr[index % n]);
    index++;
}
```

这个问题肯定还是要用单调栈的解题模板，但难点在于，比如输入是 [2,1,2,4,3]，对于最后一个元素 3，如何找到元素 4 作为 Next Greater Number。

对于这种需求，常用套路就是将数组长度翻倍：



这样，元素 3 就可以找到元素 4 作为 Next Greater Number 了，而且其他的元素都可以被正确地计算。

有了思路，最简单的实现方式当然可以把这个双倍长度的数组构造出来，然后套用算法模板。但是，**我们可以不用构造新数组，而是利用循环数组的技巧来模拟数组长度翻倍的效果**。

直接看代码吧：

```
vector<int> nextGreaterElements(vector<int>& nums) {
    int n = nums.size();
```

```
    vector<int> res(n);
    stack<int> s;
    // 假装这个数组长度翻倍了
    for (int i = 2 * n - 1; i >= 0; i--) {
        // 索引要求模，其他的和模板一样
        while (!s.empty() && s.top() <= nums[i % n]) {
            s.pop();
        }
        res[i % n] = s.empty() ? -1 : s.top();
        s.push(nums[i % n]);
    }
    return res;
}
```

这样，就可以巧妙解决环形数组的问题，时间复杂度 `O(N)`。

接下来可阅读：

- 特殊数据结构之单调队列

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



单调队列结构解决滑动窗口问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

239. 滑动窗口最大值（困难）

前文用 单调栈解决三道算法问题 介绍了单调栈这种特殊数据结构，本文写一个类似的数据结构「单调队列」。

也许这种数据结构的名字你没听过，其实没啥难的，就是一个「队列」，只是使用了一点巧妙的方法，使得**队列中的元素全都是单调递增（或递减）的**。

「单调栈」主要解决 Next Great Number 一类算法问题，而「单调队列」这个数据结构可以解决滑动窗口相关的问题，比如说力扣第 239 题「滑动窗口最大值」，难度 Hard：

给你输入一个数组 `nums` 和一个正整数 `k`，有一个大小为 `k` 的窗口在 `nums` 上从左至右滑动，请你输出每次窗口中 `k` 个元素的最大值。

函数签名如下：

```
int[] maxSlidingWindow(int[] nums, int k);
```

比如说力扣给出的一个示例：

示例:

```
输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7]
解释:

  滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

一、搭建解题框架

这道题不复杂，难点在于如何在 `O(1)` 时间算出每个「窗口」中的最大值，使得整个算法在线性时间完成。这种问题的一个特殊点在于，「窗口」是不断滑动的，也就是你得**动态地**计算窗口中的最大值。

对于这种动态的场景，很容易得到一个结论：

在一堆数字中，已知最值为 `A`，如果给这堆数添加一个数 `B`，那么比较一下 `A` 和 `B` 就可以立即算出新的最值；但如果减少一个数，就不能直接得到最值了，因为如果减少的这个数恰好是 `A`，就需要遍历所有数重新找新的最值。

回到这道题的场景，每个窗口前进的时候，要添加一个数同时减少一个数，所以想在 O(1) 的时间得出新的最值，不是那么容易的，需要「单调队列」这种特殊的数据结构来辅助。

一个普通的队列一定有这两个操作：

```
class Queue {
    // enqueue 操作，在队尾加入元素 n
    void push(int n);
    // dequeue 操作，删除队头元素
```

```java
    void pop();
}
```

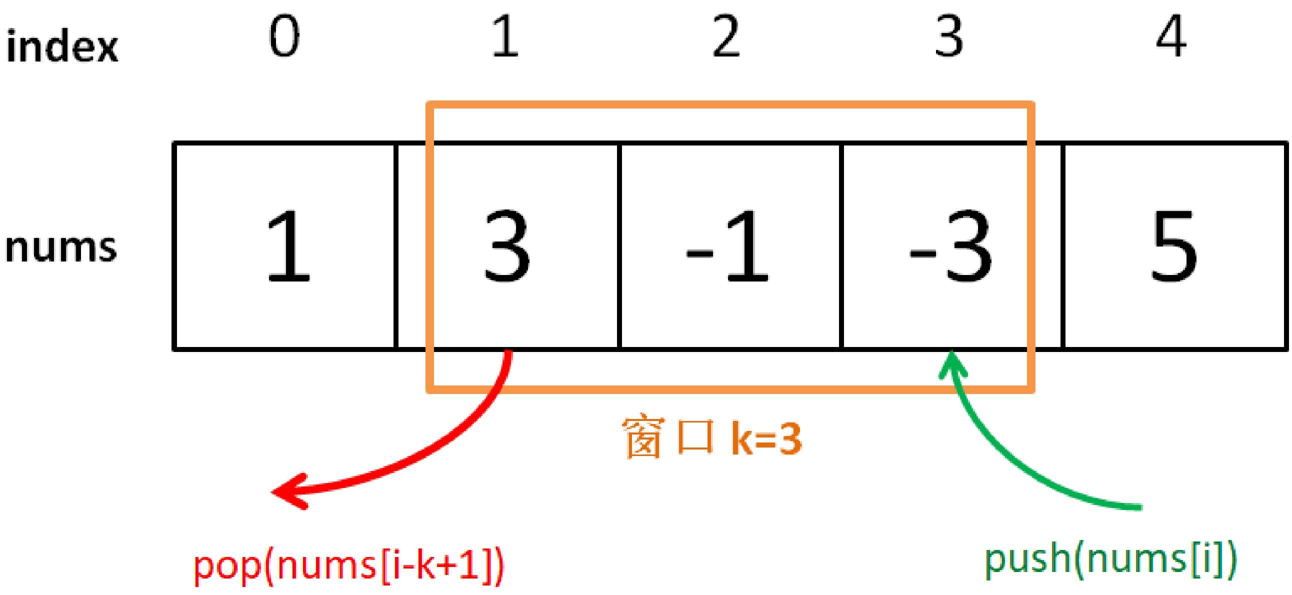
一个「单调队列」的操作也差不多：

```java
class MonotonicQueue {
    // 在队尾添加元素 n
    void push(int n);
    // 返回当前队列中的最大值
    int max();
    // 队头元素如果是 n，删除它
    void pop(int n);
}
```

当然，这几个 API 的实现方法肯定跟一般的 Queue 不一样，不过我们暂且不管，而且认为这几个操作的时间复杂度都是 O(1)，先把这道「滑动窗口」问题的解答框架搭出来：

```java
int[] maxSlidingWindow(int[] nums, int k) {
    MonotonicQueue window = new MonotonicQueue();
    List<Integer> res = new ArrayList<>();

    for (int i = 0; i < nums.length; i++) {
        if (i < k - 1) {
            //先把窗口的前 k - 1 填满
            window.push(nums[i]);
        } else {
            // 窗口开始向前滑动
            // 移入新元素
            window.push(nums[i]);
            // 将当前窗口中的最大元素记入结果
            res.add(window.max());
            // 移出最后的元素
            window.pop(nums[i - k + 1]);
        }
    }
    // 将 List 类型转化成 int[] 数组作为返回值
    int[] arr = new int[res.size()];
    for (int i = 0; i < res.size(); i++) {
        arr[i] = res.get(i);
    }
    return arr;
}
```



这个思路很简单，能理解吧？下面我们开始重头戏，单调队列的实现。

二、实现单调队列数据结构

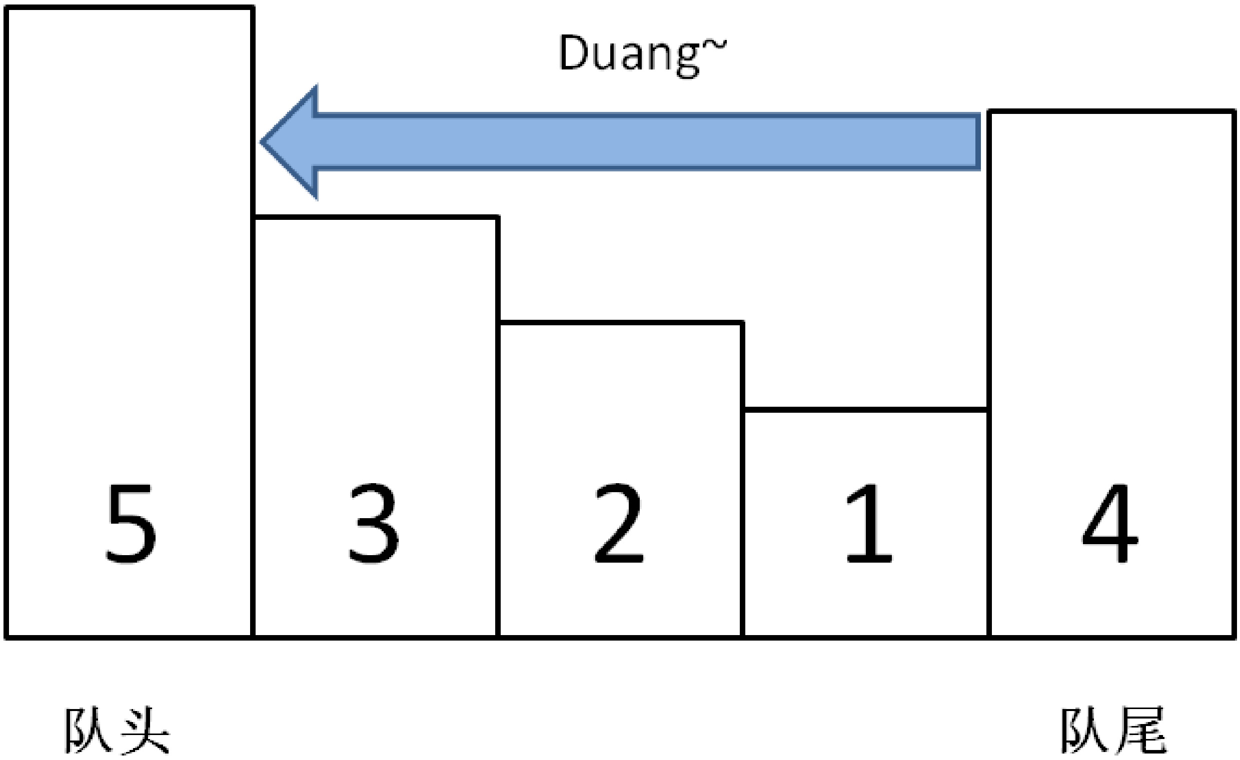
观察滑动窗口的过程就能发现，实现「单调队列」必须使用一种数据结构支持在头部和尾部进行插入和删除，很明显双链表是满足这个条件的。

「单调队列」的核心思路和「单调栈」类似，`push` 方法依然在队尾添加元素，但是要把前面比自己小的元素都删掉：

```
class MonotonicQueue {
// 双链表，支持头部和尾部增删元素
private LinkedList<Integer> q = new LinkedList<>();

public void push(int n) {
    // 将前面小于自己的元素都删除
    while (!q.isEmpty() && q.getLast() < n) {
        q.pollLast();
    }
    q.addLast(n);
}
```

你可以想象，加入数字的大小代表人的体重，把前面体重不足的都压扁了，直到遇到更大的量级才停住。



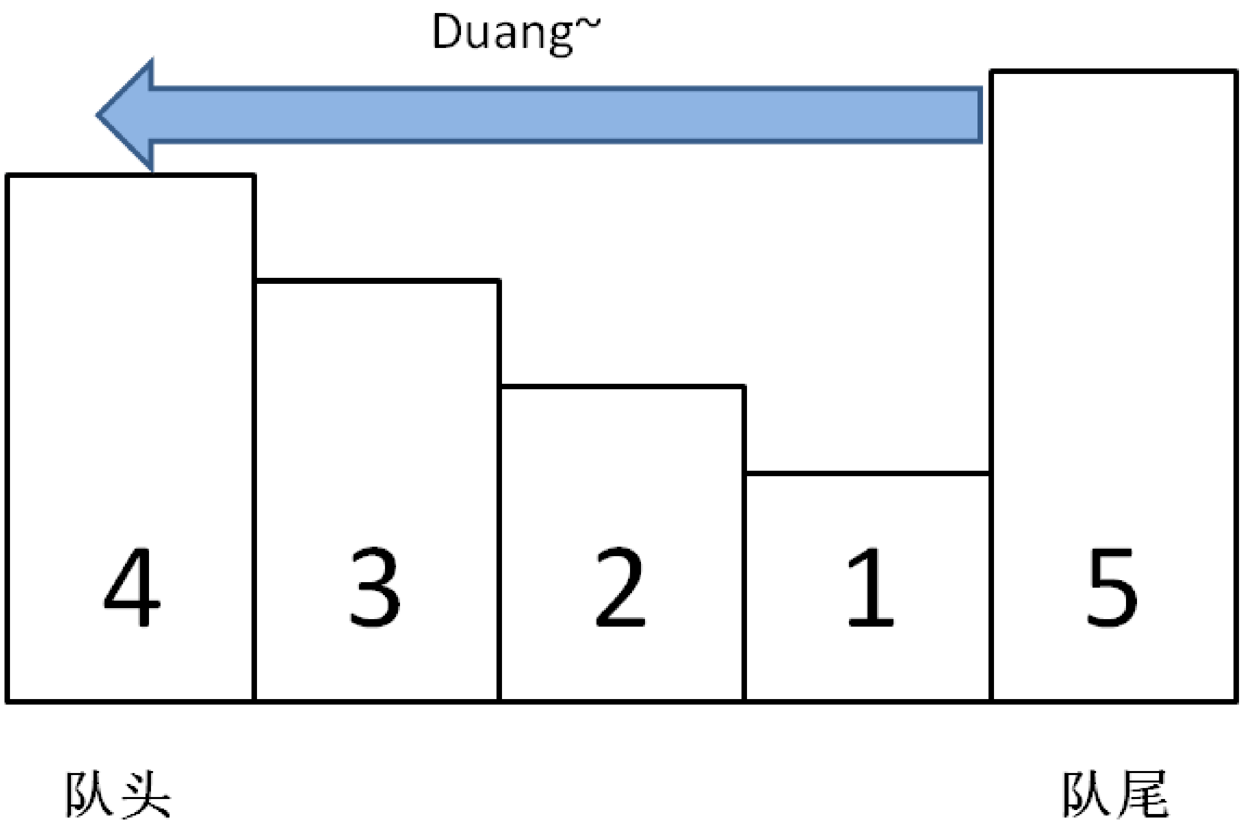
如果每个元素被加入时都这样操作，最终单调队列中的元素大小就会保持一个**单调递减**的顺序，因此我们的 max 方法可以可以这样写：

```
public int max() {
    // 队头的元素肯定是最大的
    return q.getFirst();
}
```

pop 方法在队头删除元素 n，也很好写：

```
public void pop(int n) {
    if (n == q.getFirst()) {
        q.pollFirst();
    }
}
```

之所以要判断 data.front() == n，是因为我们想删除的队头元素 n 可能已经被「压扁」了，可能已经不存在了，所以这时候就不用删除了：



至此，单调队列设计完毕，看下完整的解题代码：

```
/* 单调队列的实现 */
class MonotonicQueue {
    LinkedList<Integer> q = new LinkedList<>();
    public void push(int n) {
        // 将小于 n 的元素全部删除
        while (!q.isEmpty() && q.getLast() < n) {
            q.pollLast();
        }
        // 然后将 n 加入尾部
        q.addLast(n);
    }

    public int max() {
        return q.getFirst();
    }

    public void pop(int n) {
        if (n == q.getFirst()) {
            q.pollFirst();
        }
    }
}

/* 解题函数的实现 */
int[] maxSlidingWindow(int[] nums, int k) {
```

```java
        MonotonicQueue window = new MonotonicQueue();
        List<Integer> res = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            if (i < k - 1) {
                //先填满窗口的前 k - 1
                window.push(nums[i]);
            } else {
                // 窗口向前滑动，加入新数字
                window.push(nums[i]);
                // 记录当前窗口的最大值
                res.add(window.max());
                // 移出旧数字
                window.pop(nums[i - k + 1]);
            }
        }
        // 需要转成 int[] 数组再返回
        int[] arr = new int[res.size()];
        for (int i = 0; i < res.size(); i++) {
            arr[i] = res.get(i);
        }
        return arr;
    }
```

有一点细节问题不要忽略，在实现 `MonotonicQueue` 时，我们使用了 Java 的 `LinkedList`，因为链表结构支持在头部和尾部快速增删元素；而在解法代码中的 `res` 则使用的 `ArrayList` 结构，因为后续会按照索引取元素，所以数组结构更合适。

三、算法复杂度分析

读者可能疑惑，`push` 操作中含有 while 循环，时间复杂度应该不是 `O(1)` 呀，那么本算法的时间复杂度应该不是线性时间吧？

单独看 `push` 操作的复杂度确实不是 `O(1)`，但是算法整体的复杂度依然是 `O(N)` 线性时间。要这样想，`nums` 中的每个元素最多被 `push` 和 `pop` 一次，没有任何多余操作，所以整体的复杂度还是 `O(N)`。

空间复杂度就很简单了，就是窗口的大小 `O(k)`。

其实我觉得，这种特殊数据结构的设计还是蛮有意思的，你学会单调队列的使用了吗？学会了给个三连？

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

一道数组去重的算法题把我整不会了

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

316. 去除重复字母（中等）

1081. 不同字符的最小子序列（中等）

关于去重算法，应该没什么难度，往哈希集合里面塞不就行了么？

最多给你加点限制，问你怎么给有序数组原地去重，这个我们旧文 如何高效地给有序数组/链表去重。

本文讲的问题应该是去重相关算法中难度最大的了，把这个问题搞懂，就再也不用怕数组去重问题了。

这是力扣第 316 题「去除重复字母」，题目如下：

316. 去除重复字母 labuladong 题解 思路

难度 困难 206

给你一个仅包含小写字母的字符串，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证返回结果的字典序最小（要求不能打乱其他字符的相对位置）。

示例 1:

```
输入: "bcabc"
输出: "abc"
```

示例 2:

```
输入: "cbacdcbc"
输出: "acdb"
```

这道题和第 1081 题「不同字符的最小子序列」的解法是完全相同的，你可以把这道题的解法代码直接粘过去把 1081 题也干掉。

题目的要求总结出来有三点：

要求一、**要去重**。

要求二、去重字符串中的字符顺序**不能打乱 `s` 中字符出现的相对顺序**。

要求三、在所有符合上一条要求的去重字符串中，**字典序最小**的作为最终结果。

上述三条要求中，要求三可能有点难理解，举个例子。

比如说输入字符串 `s = "babc"`，去重且符合相对位置的字符串有两个，分别是 `"bac"` 和 `"abc"`，但是我们的算法得返回 `"abc"`，因为它的字典序更小。

按理说，如果我们想要有序的结果，那就得对原字符串排序对吧，但是排序后就不能保证符合 `s` 中字符出现顺序了，这似乎是矛盾的。

其实这里会借鉴前文 单调栈解题框架 中讲到的「单调栈」的思路，没看过也无妨，等会你就明白了。

————————————————

应合作方要求，本文不便在此发布，请扫码关注回复关键词「单调栈」查看：

1.4 数据结构设计

数据结构设计题主要就是给你提需求，让你实现 API，而且要求这些 API 的复杂度尽可能低。

根据我的经验，设计题中哈希表的出现频率很高，一般都是各类其他数据结构和哈希表组合，从而改善这些基本数据结构的特性，获得「超能力」。

公众号标签：数据结构设计

算法就像搭乐高：带你手撸 LRU 算法

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 [点这里](#)。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

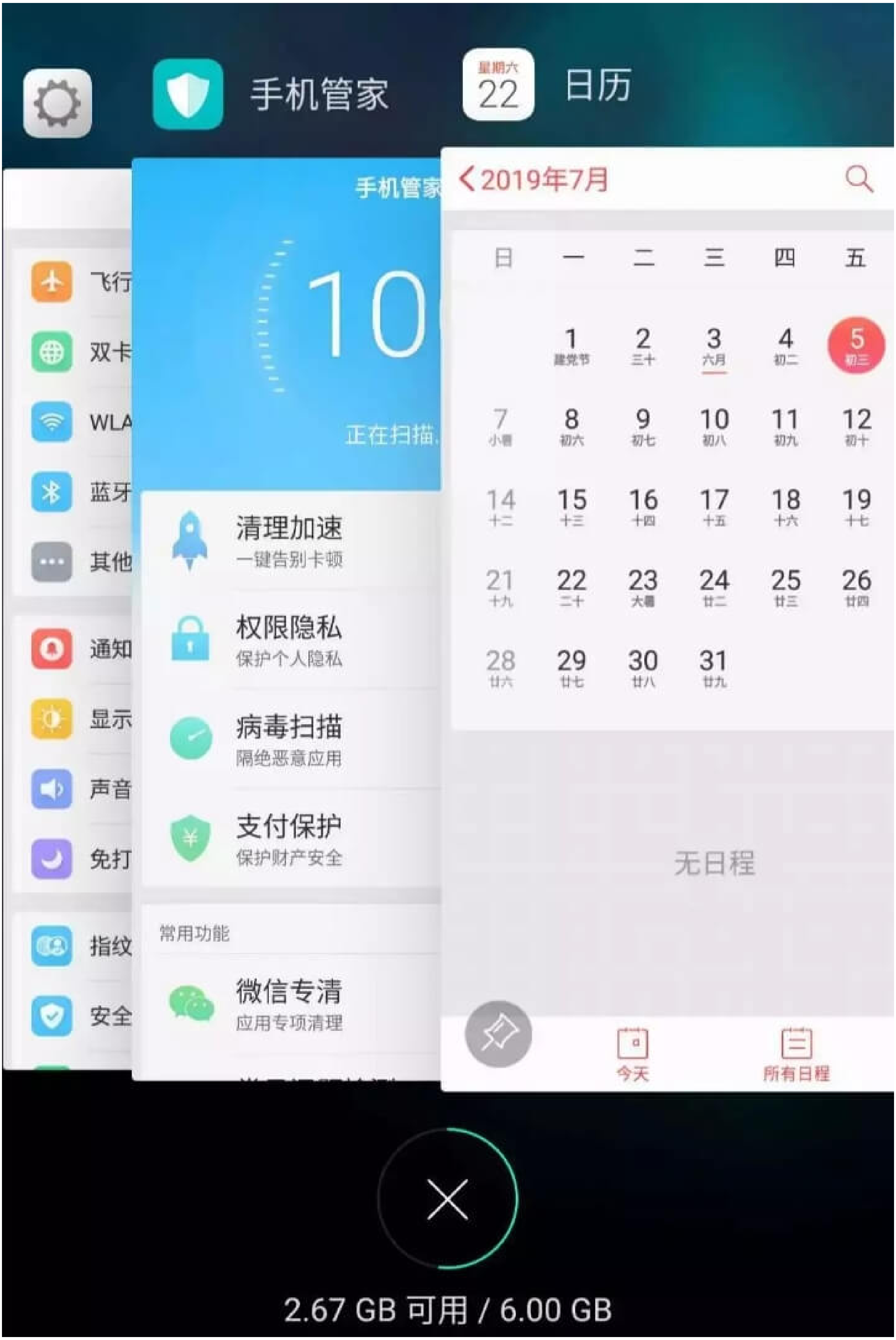
[146. LRU缓存机制（中等）](#)

LRU 算法就是一种缓存淘汰策略，原理不难，但是面试中写出没有 bug 的算法比较有技巧，需要对数据结构进行层层抽象和拆解，本文就带你写一手漂亮的代码。

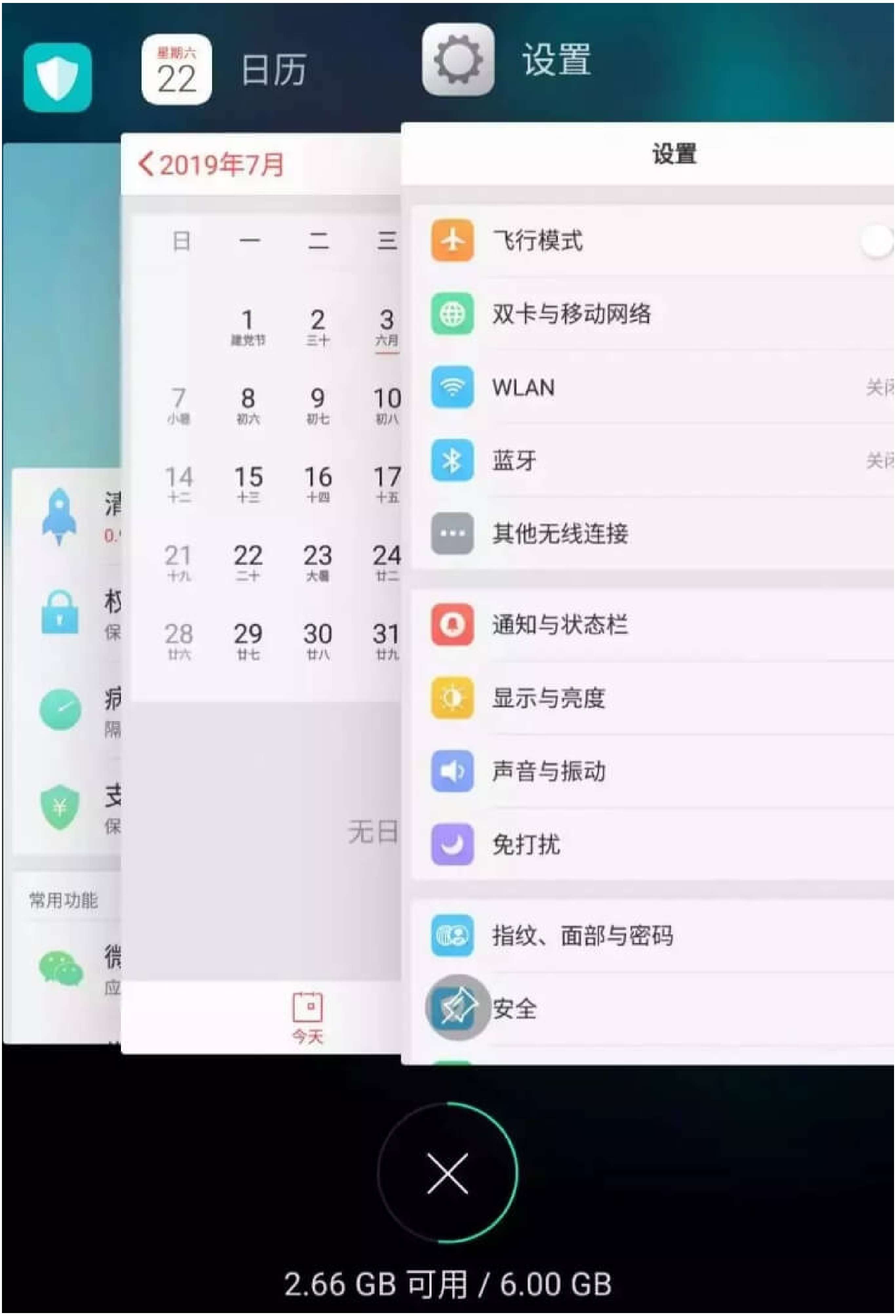
计算机的缓存容量有限，如果缓存满了就要删除一些内容，给新内容腾位置。但问题是，删除哪些内容呢？我们肯定希望删掉哪些没什么用的缓存，而把有用的数据继续留在缓存里，方便之后继续使用。那么，什么样的数据，我们判定为「有用的」的数据呢？

LRU 缓存淘汰算法就是一种常用策略。LRU 的全称是 Least Recently Used，也就是说我们认为最近使用过的数据应该是是「有用的」，很久都没用过的数据应该是无用的，内存满了就优先删那些很久没用过的数据。

举个简单的例子，安卓手机都可以把软件放到后台运行，比如我先后打开了「设置」「手机管家」「日历」，那么现在他们在后台排列的顺序是这样的：

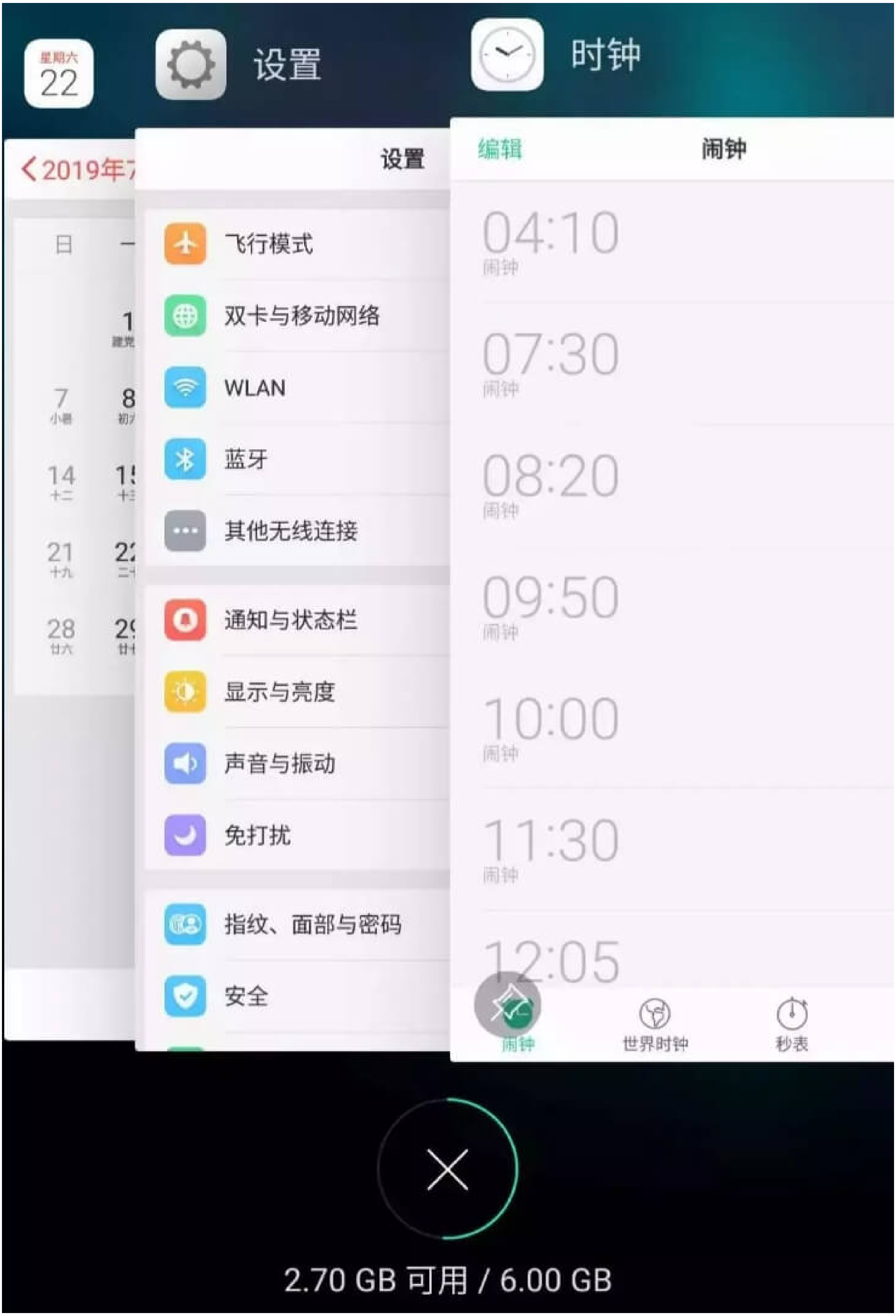

手机管家
星期六
22
日历
2019年7月
清理加速
一键告别卡顿
权限隐私
保护个人隐私
病毒扫描
隔绝恶意应用
支付保护
保护财产安全
常用功能
微信专清
应用专项清理
无日程
今天
所有日程
2.67 GB 可用 / 6.00 GB

但是这时候如果我访问了一下「设置」界面，那么「设置」就会被提前到第一个，变成这样：


日历
设置
2019年7月
设置
飞行模式
双卡与移动网络
WLAN
蓝牙
其他无线连接
通知与状态栏
显示与亮度
声音与振动
免打扰
指纹、面部与密码
安全
常用功能
今天
2.66 GB 可用 / 6.00 GB

假设我的手机只允许我同时开 3 个应用程序，现在已经满了。那么如果我新开了一个应用「时钟」，就必须关闭一个应用为「时钟」腾出一个位置，关那个呢?

按照 LRU 的策略，就关最底下的「手机管家」，因为那是最久未使用的，然后把新开的应用放到最上面：


设置
时钟
编辑
闹钟
飞行模式
双卡与移动网络
WLAN
蓝牙
其他无线连接
通知与状态栏
显示与亮度
声音与振动
免打扰
指纹、面部与密码
安全
04:10
07:30
08:20
09:50
10:00
11:30
12:05
世界时钟
秒表
2.70 GB 可用 / 6.00 GB

现在你应该理解 LRU（Least Recently Used）策略了。当然还有其他缓存淘汰策略，比如不要按访问的时序来淘汰，而是按访问频率（LFU 策略）来淘汰等等，各有应用场景。本文讲解 LRU 算法策略。

一、LRU 算法描述

力扣第 146 题「LRU缓存机制」就是让你设计数据结构：

首先要接收一个 `capacity` 参数作为缓存的最大容量，然后实现两个 API，一个是 `put(key, val)` 方法存入键值对，另一个是 `get(key)` 方法获取 `key` 对应的 `val`，如果 `key` 不存在则返回 -1。

注意哦，`get` 和 `put` 方法必须都是 `O(1)` 的时间复杂度，我们举个具体例子来看看 LRU 算法怎么工作。

```
/* 缓存容量为 2 */
LRUCache cache = new LRUCache(2);
// 你可以把 cache 理解成一个队列
// 假设左边是队头，右边是队尾
// 最近使用的排在队头，久未使用的排在队尾
// 圆括号表示键值对 (key, val)

cache.put(1, 1);
// cache = [(1, 1)]

cache.put(2, 2);
// cache = [(2, 2), (1, 1)]

cache.get(1);       // 返回 1
// cache = [(1, 1), (2, 2)]
// 解释：因为最近访问了键 1，所以提前至队头
// 返回键 1 对应的值 1

cache.put(3, 3);
// cache = [(3, 3), (1, 1)]
// 解释：缓存容量已满，需要删除内容空出位置
// 优先删除久未使用的数据，也就是队尾的数据
// 然后把新的数据插入队头

cache.get(2);       // 返回 -1 (未找到)
// cache = [(3, 3), (1, 1)]
// 解释：cache 中不存在键为 2 的数据

cache.put(1, 4);
// cache = [(1, 4), (3, 3)]
// 解释：键 1 已存在，把原始值 1 覆盖为 4
// 不要忘了也要将键值对提前到队头
```

二、LRU 算法设计

分析上面的操作过程，要让 `put` 和 `get` 方法的时间复杂度为 O(1)，我们可以总结出 `cache` 这个数据结构必要的条件：

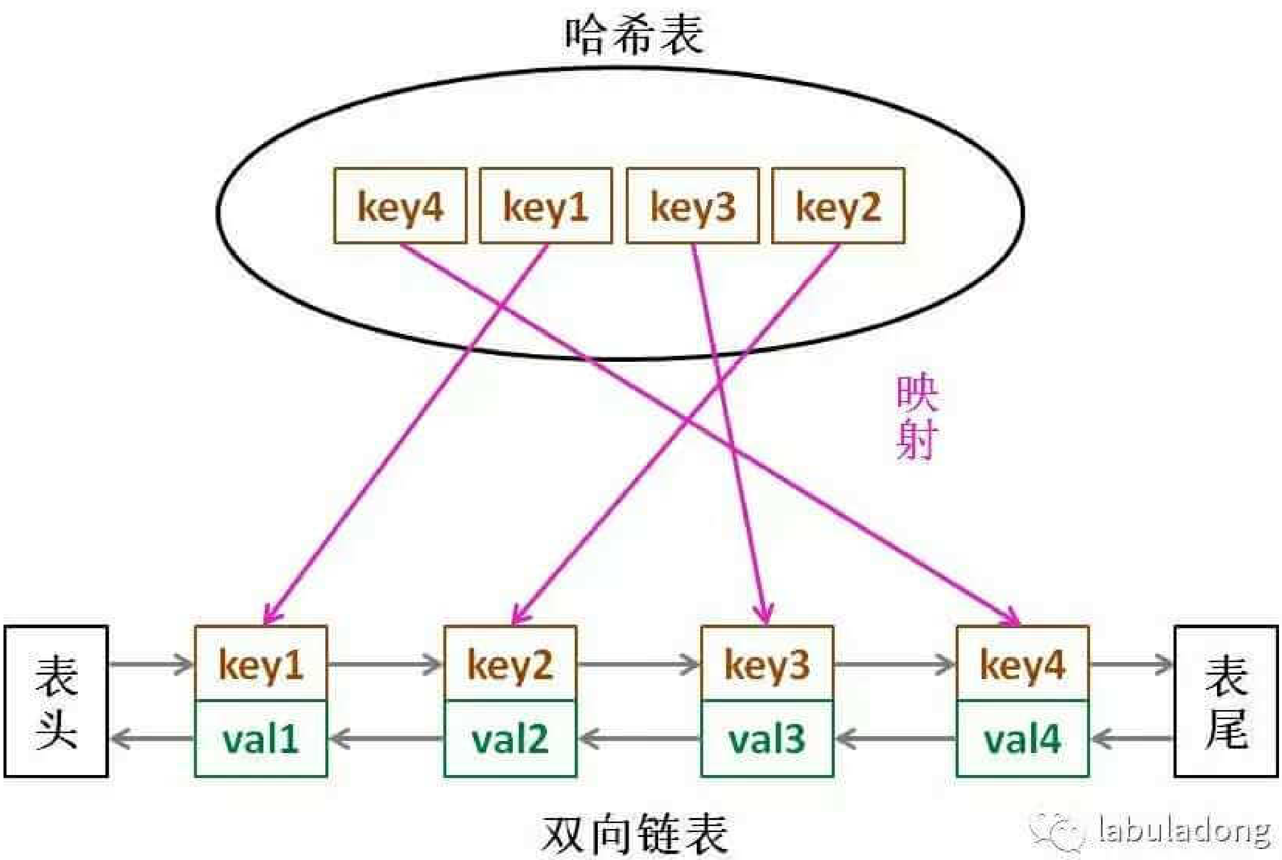
1、显然 cache 中的元素必须有时序，以区分最近使用的和久未使用的数据，当容量满了之后要删除最久未使用的那个元素腾位置。

2、我们要在 cache 中快速找某个 key 是否已存在并得到对应的 val；

3、每次访问 cache 中的某个 key，需要将这个元素变为最近使用的，也就是说 cache 要支持在任意位置快速插入和删除元素。

那么，什么数据结构同时符合上述条件呢？哈希表查找快，但是数据无固定顺序；链表有顺序之分，插入删除快，但是查找慢。所以结合一下，形成一种新的数据结构：哈希链表 LinkedHashMap。

LRU 缓存算法的核心数据结构就是哈希链表，双向链表和哈希表的结合体。这个数据结构长这样：



借助这个结构，我们来逐一分析上面的 3 个条件：

1、如果我们每次默认从链表尾部添加元素，那么显然越靠尾部的元素就是最近使用的，越靠头部的元素就是最久未使用的。

2、对于某一个 key，我们可以通过哈希表快速定位到链表中的节点，从而取得对应 val。

3、链表显然是支持在任意位置快速插入和删除的，改改指针就行。只不过传统的链表无法按照索引快速访问某一个位置的元素，而这里借助哈希表，可以通过 key 快速映射到任意一个链表节点，然后进行插入和删除。

也许读者会问，为什么要是双向链表，单链表行不行？另外，既然哈希表中已经存了 key，为什么链表中还要存 key 和 val 呢，只存 val 不就行了？

想的时候都是问题，只有做的时候才有答案。这样设计的原因，必须等我们亲自实现 LRU 算法之后才能理解，所以我们开始看代码吧～

三、代码实现

很多编程语言都有内置的哈希链表或者类似 LRU 功能的库函数，但是为了帮大家理解算法的细节，我们先自己造轮子实现一遍 LRU 算法，然后再使用 Java 内置的 `LinkedHashMap` 来实现一遍。

首先，我们把双链表的节点类写出来，为了简化，`key` 和 `val` 都认为是 int 类型：

```java
class Node {
    public int key, val;
    public Node next, prev;
    public Node(int k, int v) {
        this.key = k;
        this.val = v;
    }
}
```

然后依靠我们的 `Node` 类型构建一个双链表，实现几个 LRU 算法必须的 API：

```java
class DoubleList {
    // 头尾虚节点
    private Node head, tail;
    // 链表元素数
    private int size;

    public DoubleList() {
        // 初始化双向链表的数据
        head = new Node(0, 0);
        tail = new Node(0, 0);
        head.next = tail;
        tail.prev = head;
        size = 0;
    }

    // 在链表尾部添加节点 x，时间 O(1)
    public void addLast(Node x) {
        x.prev = tail.prev;
        x.next = tail;
        tail.prev.next = x;
        tail.prev = x;
        size++;
    }

    // 删除链表中的 x 节点（x 一定存在）
    // 由于是双链表且给的是目标 Node 节点，时间 O(1)
    public void remove(Node x) {
        x.prev.next = x.next;
        x.next.prev = x.prev;
```

```java
            size--;
        }

        // 删除链表中第一个节点，并返回该节点，时间 O(1)
        public Node removeFirst() {
            if (head.next == tail)
                return null;
            Node first = head.next;
            remove(first);
            return first;
        }

        // 返回链表长度，时间 O(1)
        public int size() { return size; }

    }
```

到这里就能回答刚才「为什么必须要用双向链表」的问题了，因为我们需要删除操作。删除一个节点不光要得到该节点本身的指针，也需要操作其前驱节点的指针，而双向链表才能支持直接查找前驱，保证操作的时间复杂度 O(1)。

注意我们实现的双链表 API 只能从尾部插入，也就是说靠尾部的数据是最近使用的，靠头部的数据是最久为使用的。

有了双向链表的实现，我们只需要在 LRU 算法中把它和哈希表结合起来即可，先搭出代码框架：

```java
class LRUCache {
    // key -> Node(key, val)
    private HashMap<Integer, Node> map;
    // Node(k1, v1) <-> Node(k2, v2)...
    private DoubleList cache;
    // 最大容量
    private int cap;

    public LRUCache(int capacity) {
        this.cap = capacity;
        map = new HashMap<>();
        cache = new DoubleList();
    }
```

先不慌去实现 LRU 算法的 `get` 和 `put` 方法。由于我们要同时维护一个双链表 `cache` 和一个哈希表 `map`，很容易漏掉一些操作，比如说删除某个 `key` 时，在 `cache` 中删除了对应的 `Node`，但是却忘记在 `map` 中删除 `key`。

解决这种问题的有效方法是：在这两种数据结构之上提供一层抽象 API。

说的有点玄幻，实际上很简单，就是尽量让 LRU 的主方法 `get` 和 `put` 避免直接操作 `map` 和 `cache` 的细节。我们可以先实现下面几个函数：

```java
/* 将某个 key 提升为最近使用的 */
private void makeRecently(int key) {
    Node x = map.get(key);
    // 先从链表中删除这个节点
    cache.remove(x);
    // 重新插到队尾
    cache.addLast(x);
}

/* 添加最近使用的元素 */
private void addRecently(int key, int val) {
    Node x = new Node(key, val);
    // 链表尾部就是最近使用的元素
    cache.addLast(x);
    // 别忘了在 map 中添加 key 的映射
    map.put(key, x);
}

/* 删除某一个 key */
private void deleteKey(int key) {
    Node x = map.get(key);
    // 从链表中删除
    cache.remove(x);
    // 从 map 中删除
    map.remove(key);
}

/* 删除最久未使用的元素 */
private void removeLeastRecently() {
    // 链表头部的第一个元素就是最久未使用的
    Node deletedNode = cache.removeFirst();
    // 同时别忘了从 map 中删除它的 key
    int deletedKey = deletedNode.key;
    map.remove(deletedKey);
}
```

这里就能回答之前的问答题「为什么要在链表中同时存储 key 和 val，而不是只存储 val」，注意 `removeLeastRecently` 函数中，我们需要用 `deletedNode` 得到 `deletedKey`。

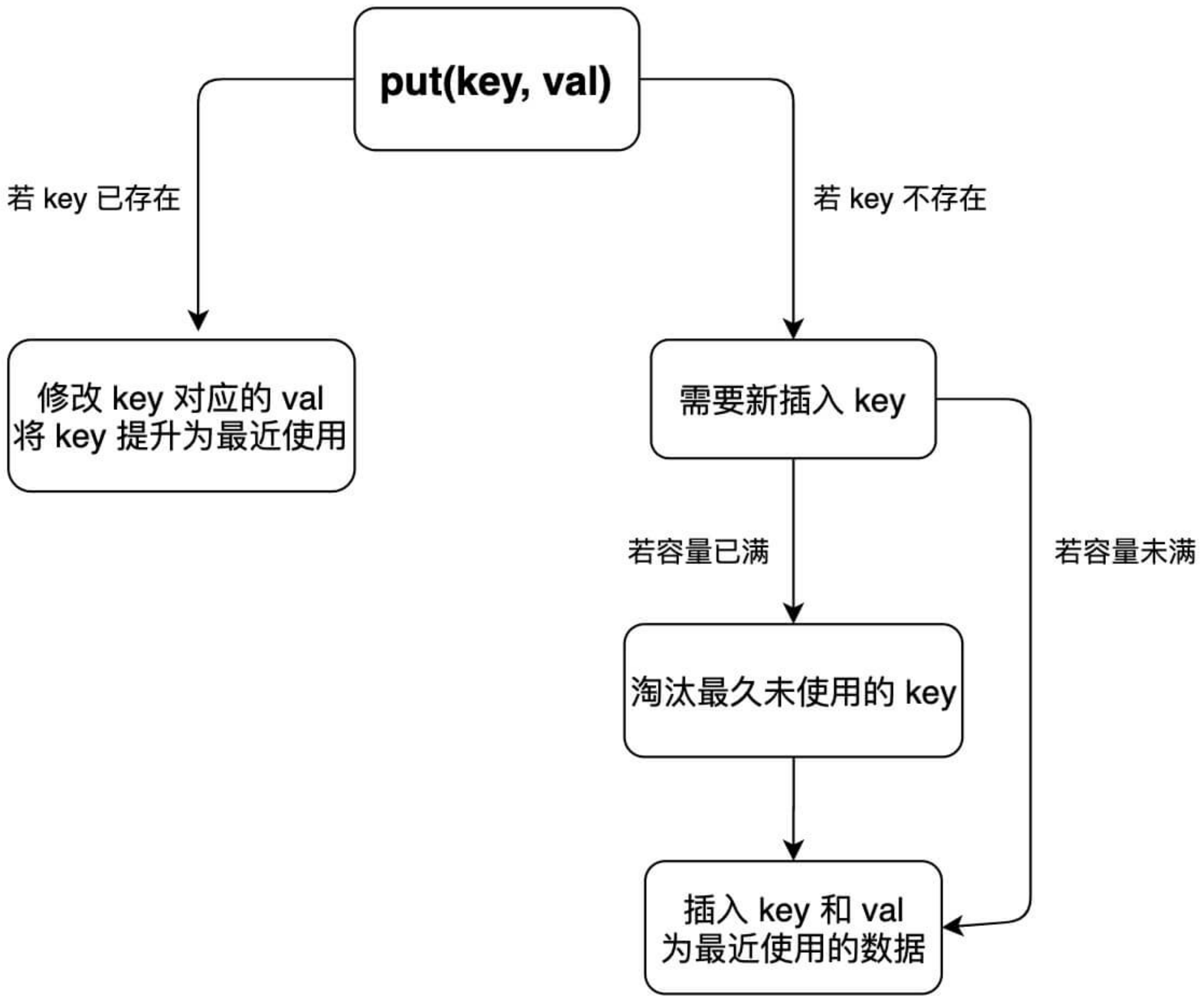
也就是说，当缓存容量已满，我们不仅仅要删除最后一个 `Node` 节点，还要把 `map` 中映射到该节点的 `key` 同时删除，而这个 `key` 只能由 `Node` 得到。如果 `Node` 结构中只存储 `val`，那么我们就无法得知 `key` 是什么，就无法删除 `map` 中的键，造成错误。

上述方法就是简单的操作封装，调用这些函数可以避免直接操作 `cache` 链表和 `map` 哈希表，下面我先来实现 LRU 算法的 `get` 方法：

```java
public int get(int key) {
    if (!map.containsKey(key)) {
        return -1;
    }
```

```
        // 将该数据提升为最近使用的
        makeRecently(key);
        return map.get(key).val;
    }
```

put 方法稍微复杂一些，我们先来画个图搞清楚它的逻辑：



这样我们可以轻松写出 put 方法的代码：

```
public void put(int key, int val) {
    if (map.containsKey(key)) {
        // 删除旧的数据
        deleteKey(key);
        // 新插入的数据为最近使用的数据
        addRecently(key, val);
        return;
    }

    if (cap == cache.size()) {
        // 删除最久未使用的元素
```

```
            removeLeastRecently();
        }
        // 添加为最近使用的元素
        addRecently(key, val);
    }
```

至此，你应该已经完全掌握 LRU 算法的原理和实现了，我们最后用 Java 的内置类型 `LinkedHashMap` 来实现 LRU 算法，逻辑和之前完全一致，我就不过多解释了：

```
class LRUCache {
    int cap;
    LinkedHashMap<Integer, Integer> cache = new LinkedHashMap<>();
    public LRUCache(int capacity) {
        this.cap = capacity;
    }

    public int get(int key) {
        if (!cache.containsKey(key)) {
            return -1;
        }
        // 将 key 变为最近使用
        makeRecently(key);
        return cache.get(key);
    }

    public void put(int key, int val) {
        if (cache.containsKey(key)) {
            // 修改 key 的值
            cache.put(key, val);
            // 将 key 变为最近使用
            makeRecently(key);
            return;
        }

        if (cache.size() >= this.cap) {
            // 链表头部就是最久未使用的 key
            int oldestKey = cache.keySet().iterator().next();
            cache.remove(oldestKey);
        }
        // 将新的 key 添加链表尾部
        cache.put(key, val);
    }

    private void makeRecently(int key) {
        int val = cache.get(key);
        // 删除 key，重新插入到队尾
        cache.remove(key);
        cache.put(key, val);
    }
}
```

至此，LRU 算法就没有什么神秘的了。

接下来可阅读：

- 手把手带你实现 LFU 算法

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



算法就像搭乐高：带你手撸 LFU 算法

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

460. LFU 缓存机制（困难）

上篇文章 带你手写LRU算法 写了 LRU 缓存淘汰算法的实现方法，本文来写另一个著名的缓存淘汰算法：LFU 算法。

LRU 算法的淘汰策略是 Least Recently Used，也就是每次淘汰那些最久没被使用的数据；而 LFU 算法的淘汰策略是 Least Frequently Used，也就是每次淘汰那些使用次数最少的数据。

LRU 算法的核心数据结构是使用哈希链表 `LinkedHashMap`，首先借助链表的有序性使得链表元素维持插入顺序，同时借助哈希映射的快速访问能力使得我们可以在 O(1) 时间访问链表的任意元素。

从实现难度上来说，LFU 算法的难度大于 LRU 算法，因为 LRU 算法相当于把数据按照时间排序，这个需求借助链表很自然就能实现，你一直从链表头部加入元素的话，越靠近头部的元素就是新的数据，越靠近尾部的元素就是旧的数据，我们进行缓存淘汰的时候只要简单地将尾部的元素淘汰掉就行了。

而 LFU 算法相当于是把数据按照访问频次进行排序，这个需求恐怕没有那么简单，而且还有一种情况，如果多个数据拥有相同的访问频次，我们就得删除最早插入的那个数据。也就是说 LFU 算法是淘汰访问频次最低的数据，如果访问频次最低的数据有多条，需要淘汰最旧的数据。

所以说 LFU 算法是要复杂很多的，而且经常出现在面试中，因为 LFU 缓存淘汰算法在工程实践中经常使用，也有可能是应该 LRU 算法太简单了。**不过话说回来，这种著名的算法的套路都是固定的，关键是由于逻辑较复杂，不容易写出漂亮且没有 bug 的代码。**

那么本文我就带你拆解 LFU 算法，自顶向下，逐步求精，就是解决复杂问题的不二法门。

一、算法描述

要求你写一个类，接受一个 `capacity` 参数，实现 `get` 和 `put` 方法：

```
class LFUCache {
    // 构造容量为 capacity 的缓存
    public LFUCache(int capacity) {}
    // 在缓存中查询 key
    public int get(int key) {}
```

```
    // 将 key 和 val 存入缓存
    public void put(int key, int val) {}
}
```

`get(key)` 方法会去缓存中查询键 `key`，如果 `key` 存在，则返回 `key` 对应的 `val`，否则返回 -1。

`put(key, value)` 方法插入或修改缓存。如果 `key` 已存在，则将它对应的值改为 `val`；如果 `key` 不存在，则插入键值对 `(key, val)`。

当缓存达到容量 `capacity` 时，则应该在插入新的键值对之前，删除使用频次（后文用 `freq` 表示）最低的键值对。如果 `freq` 最低的键值对有多个，则删除其中最旧的那个。

```java
// 构造一个容量为 2 的 LFU 缓存
LFUCache cache = new LFUCache(2);

// 插入两对 (key, val)，对应的 freq 为 1
cache.put(1, 10);
cache.put(2, 20);

// 查询 key 为 1 对应的 val
// 返回 10，同时键 1 对应的 freq 变为 2
cache.get(1);

// 容量已满，淘汰 freq 最小的键 2
// 插入键值对 (3, 30)，对应的 freq 为 1
cache.put(3, 30);

// 键 2 已经被淘汰删除，返回 -1
cache.get(2);
```

二、思路分析

一定先从最简单的开始，根据 LFU 算法的逻辑，我们先列举出算法执行过程中的几个显而易见的事实：

1、调用 `get(key)` 方法时，要返回该 `key` 对应的 `val`。

2、只要用 `get` 或者 `put` 方法访问一次某个 `key`，该 `key` 的 `freq` 就要加一。

3、如果在容量满了的时候进行插入，则需要将 `freq` 最小的 `key` 删除，如果最小的 `freq` 对应多个 `key`，则删除其中最旧的那一个。

好的，我们希望能够在 O(1) 的时间内解决这些需求，可以使用基本数据结构来逐个击破：

1、使用一个 `HashMap` 存储 `key` 到 `val` 的映射，就可以快速计算 `get(key)`。

```java
HashMap<Integer, Integer> keyToVal;
```

2、使用一个 `HashMap` 存储 `key` 到 `freq` 的映射，就可以快速操作 `key` 对应的 `freq`。

```
HashMap<Integer, Integer> keyToFreq;
```

3、这个需求应该是 LFU 算法的核心，所以我们分开说。

3.1、首先，肯定是需要 `freq` 到 `key` 的映射，用来找到 `freq` 最小的 `key`。

3.2、将 `freq` 最小的 `key` 删除，那你就得快速得到当前所有 `key` 最小的 `freq` 是多少。想要时间复杂度 O(1) 的话，肯定不能遍历一遍去找，那就用一个变量 `minFreq` 来记录当前最小的 `freq` 吧。

3.3、可能有多个 `key` 拥有相同的 `freq`，所以 **`freq` 对 `key` 是一对多的关系**，即一个 `freq` 对应一个 `key` 的列表。

3.4、希望 `freq` 对应的 `key` 的列表是**存在时序**的，便于快速查找并删除最旧的 `key`。

3.5、希望**能够快速删除 `key` 列表中的任何一个 `key`**，因为如果频次为 `freq` 的某个 `key` 被访问，那么它的频次就会变成 `freq+1`，就应该从 `freq` 对应的 `key` 列表中删除，加到 `freq+1` 对应的 `key` 的列表中。

```
HashMap<Integer, LinkedHashSet<Integer>> freqToKeys;
int minFreq = 0;
```

介绍一下这个 `LinkedHashSet`，它满足我们 3.3，3.4，3.5 这几个要求。你会发现普通的链表 `LinkedList` 能够满足 3.3，3.4 这两个要求，但是由于普通链表不能快速访问链表中的某一个节点，所以无法满足 3.5 的要求。

`LinkedHashSet` 顾名思义，是链表和哈希集合的结合体。链表不能快速访问链表节点，但是插入元素具有时序；哈希集合中的元素无序，但是可以对元素进行快速的访问和删除。

那么，它俩结合起来就兼具了哈希集合和链表的特性，既可以在 O(1) 时间内访问或删除其中的元素，又可以保持插入的时序，高效实现 3.5 这个需求。

综上，我们可以写出 LFU 算法的基本数据结构：

```
class LFUCache {
    // key 到 val 的映射，我们后文称为 KV 表
    HashMap<Integer, Integer> keyToVal;
    // key 到 freq 的映射，我们后文称为 KF 表
    HashMap<Integer, Integer> keyToFreq;
    // freq 到 key 列表的映射，我们后文称为 FK 表
    HashMap<Integer, LinkedHashSet<Integer>> freqToKeys;
    // 记录最小的频次
    int minFreq;
    // 记录 LFU 缓存的最大容量
    int cap;

    public LFUCache(int capacity) {
        keyToVal = new HashMap<>();
        keyToFreq = new HashMap<>();
        freqToKeys = new HashMap<>();
```

```
        this.cap = capacity;
        this.minFreq = 0;
    }

    public int get(int key) {}

    public void put(int key, int val) {}

}
```

三、代码框架

LFU 的逻辑不难理解，但是写代码实现并不容易，因为你看我们要维护 KV 表，KF 表，FK 表三个映射，特别容易出错。对于这种情况，labuladong 教你三个技巧：

1、不要企图上来就实现算法的所有细节，而应该自顶向下，逐步求精，先写清楚主函数的逻辑框架，然后再一步步实现细节。

2、搞清楚映射关系，如果我们更新了某个 `key` 对应的 `freq`，那么就要同步修改 KF 表和 FK 表，这样才不会出问题。

3、画图，画图，画图，重要的话说三遍，把逻辑比较复杂的部分用流程图画出来，然后根据图来写代码，可以极大减少出错的概率。

下面我们先来实现 `get(key)` 方法，逻辑很简单，返回 `key` 对应的 `val`，然后增加 `key` 对应的 `freq`：

应合作方要求，本文不便在此发布，请扫码关注回复关键词「lfu」查看：

给我常数时间，我可以删除/查找数组中的任意元素

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

380. 常数时间插入、删除和获取随机元素（中等）

710. 黑名单中的随机数（困难）

本文讲两道比较有技巧性的数据结构设计题，都是和随机读取元素相关的，我们前文 水塘抽样算法 也写过类似的问题。

这写问题的一个技巧点在于，如何结合哈希表和数组，使得数组的删除操作时间复杂度也变成 O(1)?

下面来一道道看。

实现随机集合

这是力扣第 380 题「常数时间插入、删除和获取随机元素」，看下题目：

380. 常数时间插入、删除和获取随机元素 labuladong 题解 思路

难度 中等 192 收藏 分享 切换为英文 关注 反馈

设计一个支持在*平均*时间复杂度 **O(1)** 下，执行以下操作的数据结构。

1. `insert(val)`：当元素 val 不存在时，向集合中插入该项。
2. `remove(val)`：元素 val 存在时，从集合中移除该项。
3. `getRandom`：随机返回现有集合中的一项。每个元素应该有**相同的概率**被返回。

示例：

```
// 初始化一个空的集合。
RandomizedSet randomSet = new RandomizedSet();

// 向集合中插入 1 。返回 true 表示 1 被成功地插入。
randomSet.insert(1);

// 返回 false ，表示集合中不存在 2 。
randomSet.remove(2);

// 向集合中插入 2 。返回 true 。集合现在包含 [1,2] 。
randomSet.insert(2);

// getRandom 应随机返回 1 或 2 。
randomSet.getRandom();

// 从集合中移除 1 ，返回 true 。集合现在包含 [2] 。
randomSet.remove(1);
```

就是说就是让我们实现如下一个类：

```
class RandomizedSet {
public:
    /** 如果 val 不存在集合中，则插入并返回 true，否则直接返回 false */
     bool insert(int val) {}

    /** 如果 val 在集合中，则删除并返回 true，否则直接返回 false */
    bool remove(int val) {}

    /** 从集合中等概率地随机获得一个元素 */
    int getRandom() {}
}
```

本题的难点在于两点：

1、插入，删除，获取随机元素这三个操作的时间复杂度必须都是 O(1)。

2、`getRandom` 方法返回的元素必须等概率返回随机元素，也就是说，如果集合里面有 `n` 个元素，每个元素被返回的概率必须是 `1/n`。

我们先来分析一下：对于插入，删除，查找这几个操作，哪种数据结构的时间复杂度是 O(1)?

`HashSet` 肯定算一个对吧。哈希集合的底层原理就是一个大数组，我们把元素通过哈希函数映射到一个索引上；如果用拉链法解决哈希冲突，那么这个索引可能连着一个链表或者红黑树。

那么请问对于这样一个标准的 `HashSet`，你能否在 O(1) 的时间内实现 `getRandom` 函数?

其实是不能的，因为根据刚才说到的底层实现，元素是被哈希函数「分散」到整个数组里面的，更别说还有拉链法等等解决哈希冲突的机制，基本做不到 O(1) 时间等概率随机获取元素。

除了 `HashSet`，还有一些类似的数据结构，比如哈希链表 `LinkedHashSet`，我们前文 手把手实现LRU算法 和 手把手实现LFU算法 讲过这类数据结构的实现原理，本质上就是哈希表配合双链表，元素存储在双链表中。

但是，`LinkedHashSet` 只是给 `HashSet` 增加了有序性，依然无法按要求实现我们的 `getRandom` 函数，因为底层用链表结构存储元素的话，是无法在 O(1) 的时间内访问某一个元素的。

根据上面的分析，对于 `getRandom` 方法，如果想「等概率」且「在 O(1) 的时间」取出元素，一定要满足：**底层用数组实现，且数组必须是紧凑的。**

这样我们就可以直接生成随机数作为索引，从数组中取出该随机索引对应的元素，作为随机元素。

但如果用数组存储元素的话，插入，删除的时间复杂度怎么可能是 O(1) 呢?

可以做到！对数组尾部进行插入和删除操作不会涉及数据搬移，时间复杂度是 O(1)。

所以，如果我们想在 O(1) 的时间删除数组中的某一个元素 `val`，可以先把这个元素交换到数组的尾部，然后再 `pop` 掉。

交换两个元素必须通过索引进行交换对吧，那么我们需要一个哈希表 `valToIndex` 来记录每个元素值对应的索引。

有了思路铺垫，我们直接看代码：

```
class RandomizedSet {
public:
    // 存储元素的值
    vector<int> nums;
    // 记录每个元素对应在 nums 中的索引
    unordered_map<int,int> valToIndex;

    bool insert(int val) {
        // 若 val 已存在，不用再插入
        if (valToIndex.count(val)) {
```

```
            return false;
        }
        // 若 val 不存在，插入到 nums 尾部，
        // 并记录 val 对应的索引值
        valToIndex[val] = nums.size();
        nums.push_back(val);
        return true;
    }

    bool remove(int val) {
        // 若 val 不存在，不用再删除
        if (!valToIndex.count(val)) {
            return false;
        }
        // 先拿到 val 的索引
        int index = valToIndex[val];
        // 将最后一个元素对应的索引修改为 index
        valToIndex[nums.back()] = index;
        // 交换 val 和最后一个元素
        swap(nums[index], nums.back());
        // 在数组中删除元素 val
        nums.pop_back();
        // 删除元素 val 对应的索引
        valToIndex.erase(val);
        return true;
    }

    int getRandom() {
        // 随机获取 nums 中的一个元素
        return nums[rand() % nums.size()];
    }
};
```

注意 `remove(val)` 函数，对 `nums` 进行插入、删除、交换时，都要记得修改哈希表 `valToIndex`，否则会出现错误。

至此，这道题就解决了，每个操作的复杂度都是 O(1)，且随机抽取的元素概率是相等的。

避开黑名单的随机数

有了上面一道题的铺垫，我们来看一道更难一些的题目，力扣第 710 题「黑名单中的随机数」，我来描述一下题目：

给你输入一个正整数 `N`，代表左闭右开区间 `[0,N)`，再给你输入一个数组 `blacklist`，其中包含一些「黑名单数字」，且 `blacklist` 中的数字都是区间 `[0,N)` 中的数字。

现在要求你设计如下数据结构：

```
class Solution {
public:
    // 构造函数，输入参数
```

```
    Solution(int N, vector<int>& blacklist) {}

    // 在区间 [0,N) 中等概率随机选取一个元素并返回
    // 这个元素不能是 blacklist 中的元素
    int pick() {}
};
```

pick 函数会被多次调用，每次调用都要在区间 [0,N) 中「等概率随机」返回一个「不在 blacklist 中」的整数。

这应该不难理解吧，比如给你输入 N = 5, blacklist = [1,3]，那么多次调用 pick 函数，会等概率随机返回 0, 2, 4 中的某一个数字。

而且题目要求，在 pick 函数中应该尽可能少调用随机数生成函数 rand()。

这句话什么意思呢，比如说我们可能想出如下拍脑袋的解法：

```
int pick() {
    int res = rand() % N;
    while (res exists in blacklist) {
        // 重新随机一个结果
        res = rand() % N;
    }
    return res;
}
```

这个函数会多次调用 rand() 函数，执行效率竟然和随机数相关，不是一个漂亮的解法。

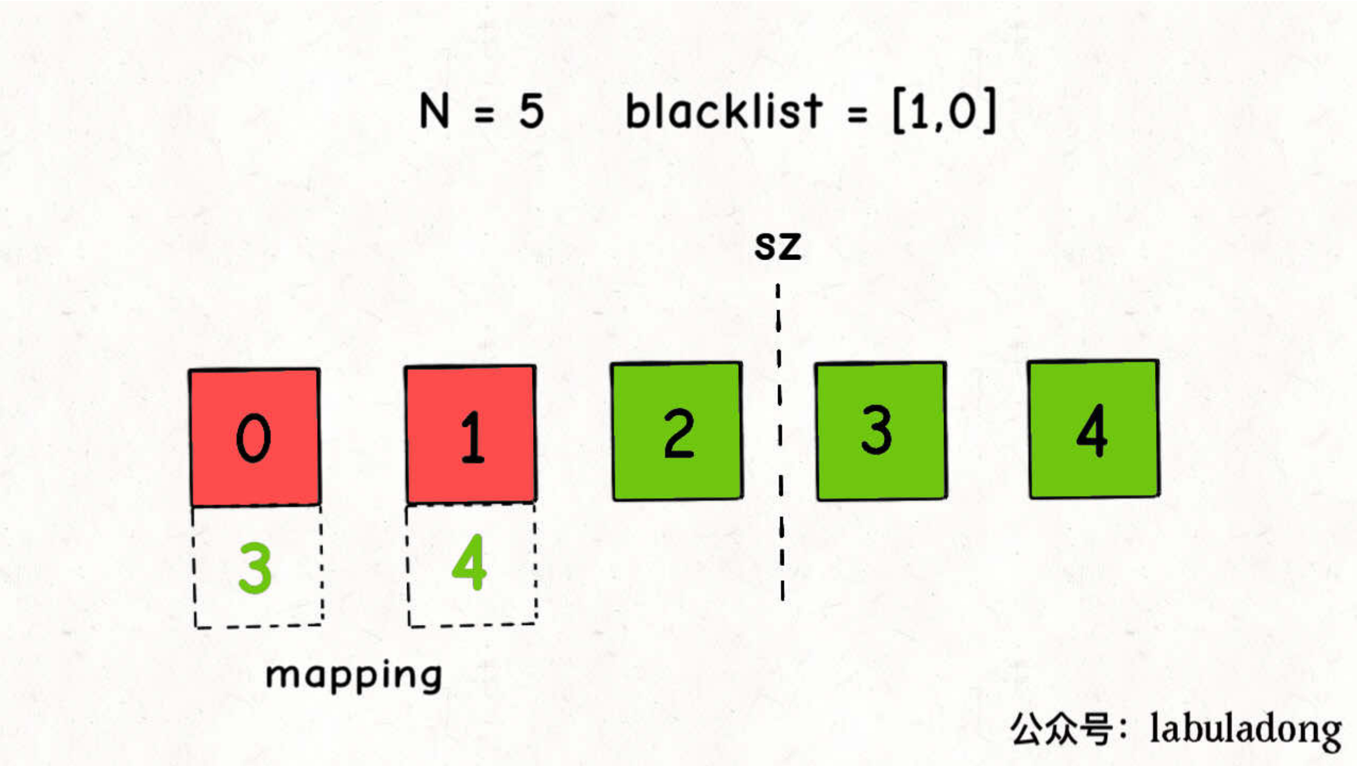
聪明的解法类似上一道题，我们可以将区间 [0,N) 看做一个数组，然后将 blacklist 中的元素移到数组的最末尾，同时用一个哈希表进行映射：

根据这个思路，我们可以写出第一版代码（还存在几处错误）：

```
class Solution {
public:
    int sz;
    unordered_map<int, int> mapping;

    Solution(int N, vector<int>& blacklist) {
        // 最终数组中的元素个数
        sz = N - blacklist.size();
        // 最后一个元素的索引
        int last = N - 1;
        // 将黑名单中的索引换到最后去
        for (int b : blacklist) {
            mapping[b] = last;
            last--;
        }
```

```
    }
};
```



如上图，相当于把黑名单中的数字都交换到了区间 `[sz, N)` 中，同时把 `[0, sz)` 中的黑名单数字映射到了正常数字。

根据这个逻辑，我们可以写出 `pick` 函数：

```
int pick() {
    // 随机选取一个索引
    int index = rand() % sz;
    // 这个索引命中了黑名单，
    // 需要被映射到其他位置
    if (mapping.count(index)) {
        return mapping[index];
    }
    // 若没命中黑名单，则直接返回
    return index;
}
```

这个 `pick` 函数已经没有问题了，但是构造函数还有两个问题。

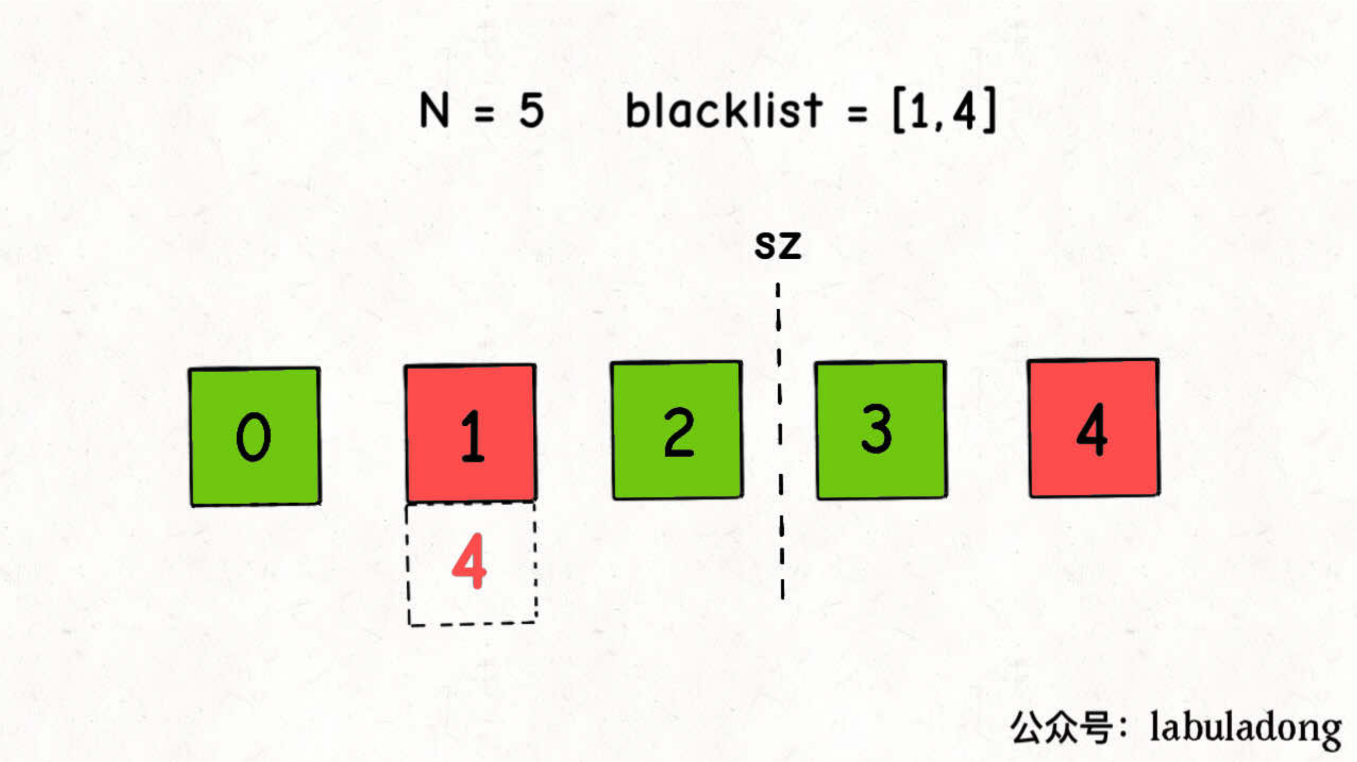
第一个问题，如下这段代码：

```
int last = N - 1;
// 将黑名单中的索引换到最后去
for (int b : blacklist) {
    mapping[b] = last;
```

```
        last--;
    }
```

我们将黑名单中的 `b` 映射到 `last`，但是我们能确定 `last` 不在 `blacklist` 中吗？

比如下图这种情况，我们的预期应该是 1 映射到 3，但是错误地映射到 4：

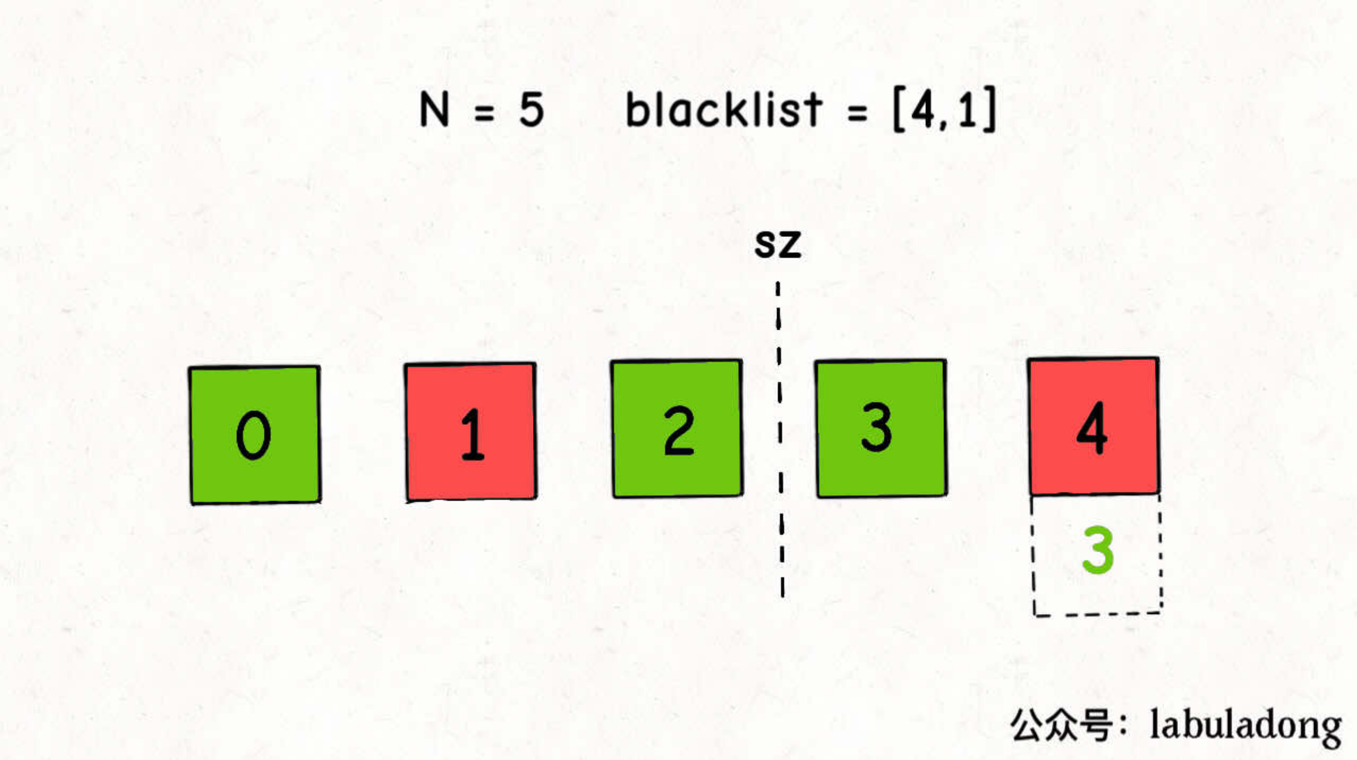


在对 `mapping[b]` 赋值时，要保证 `last` 一定不在 `blacklist` 中，可以如下操作：

```
// 构造函数
Solution(int N, vector<int>& blacklist) {
    sz = N - blacklist.size();
    // 先将所有黑名单数字加入 map
    for (int b : blacklist) {
        // 这里赋值多少都可以
        // 目的仅仅是把键存进哈希表
        // 方便快速判断数字是否在黑名单内
        mapping[b] = 666;
    }

    int last = N - 1;
    for (int b : blacklist) {
        // 跳过所有黑名单中的数字
        while (mapping.count(last)) {
            last--;
        }
        // 将黑名单中的索引映射到合法数字
        mapping[b] = last;
        last--;
    }
}
```

第二个问题，如果 `blacklist` 中的黑名单数字本身就存在区间 `[sz, N)` 中，那么就没必要在 `mapping` 中建立映射，比如这种情况：



我们根本不用管 4，只希望把 1 映射到 3，但是按照 `blacklist` 的顺序，会把 4 映射到 3，显然是错误的。

我们可以稍微修改一下，写出正确的解法代码：

```
class Solution {
public:
    int sz;
    unordered_map<int, int> mapping;

    Solution(int N, vector<int>& blacklist) {
        sz = N - blacklist.size();
        for (int b : blacklist) {
            mapping[b] = 666;
        }

        int last = N - 1;
        for (int b : blacklist) {
            // 如果 b 已经在区间 [sz, N)
            // 可以直接忽略
            if (b >= sz) {
                continue;
            }
            while (mapping.count(last)) {
                last--;
            }
            mapping[b] = last;
```

```
                last--;
            }
        }

        // 见上文代码实现
        int pick() {}
};
```

至此，这道题也解决了，总结一下本文的核心思想：

1、如果想高效地，等概率地随机获取元素，就要使用数组作为底层容器。

2、如果要保持数组元素的紧凑性，可以把待删除元素换到最后，然后 pop 掉末尾的元素，这样时间复杂度就是 O(1) 了。当然，我们需要额外的哈希表记录值到索引的映射。

3、对于第二题，数组中含有「空洞」（黑名单数字），也可以利用哈希表巧妙处理映射关系，让数组在逻辑上是紧凑的，方便随机取元素。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



一道求中位数的算法题把我整不会了

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

295. 数据流的中位数（困难）

如果输入一个数组，让你求中位数，这个好办，排个序，如果数组长度是奇数，最中间的一个元素就是中位数，如果数组长度是偶数，最中间两个元素的平均数作为中位数。

如果数据规模非常巨大，排序不太现实，那么也可以使用概率算法，随机抽取一部分数据，排序，求中位数，作为所有数据的中位数。

本文说的中位数算法比较困难，也比较精妙，是力扣第 295 题「数据流的中位数」：

295. 数据流的中位数

难度 困难 251

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

设计一个支持以下两种操作的数据结构：

- void addNum(int num) - 从数据流中添加一个整数到数据结构中。
- double findMedian() - 返回目前所有元素的中位数。

示例：

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

就是让你设计这样一个类：

```java
class MedianFinder {

    // 添加一个数字
    public void addNum(int num) {}

    // 计算当前添加的所有数字的中位数
    public double findMedian() {}
}
```

其实，所有关于「流」的算法都比较难，比如我们旧文 水塘抽样算法详解 写过如何从数据流中等概率随机抽取一个元素，如果说你没有接触过这个问题的话，还是很难想到解法的。

这道题要求在数据流中计算平均数，我们先想一想常规思路。

尝试分析

一个直接的解法可以用一个数组记录所有 `addNum` 添加进来的数字，通过插入排序的逻辑保证数组中的元素有序，当调用 `findMedian` 方法时，可以通过数组索引直接计算中位数。

但是用数组作为底层容器的问题也很明显，`addNum` 搜索插入位置的时候可以用二分搜索算法，但是插入操作需要搬移数据，所以最坏时间复杂度为 O(N)。

那换链表？链表插入元素很快，但是查找插入位置的时候只能线性遍历，最坏时间复杂度还是 O(N)，而且 `findMedian` 方法也需要遍历寻找中间索引，最坏时间复杂度也是 O(N)。

那么就用平衡二叉树呗，增删查改复杂度都是 O(logN)，这样总行了吧？

比如用 Java 提供的 `TreeSet` 容器，底层是红黑树，`addNum` 直接插入，`findMedian` 可以通过当前元素的个数推出计算中位数的元素的排名。

很遗憾，依然不行，这里有两个问题。

第一，`TreeSet` 是一种 `Set`，其中不存在重复元素的元素，但是我们的数据流可能输入重复数据的，而且计算中位数也是需要算上重复元素的。

第二，`TreeSet` 并没有实现一个通过排名快速计算元素的 API。假设我想找到 `TreeSet` 中第 5 大的元素，并没有一个现成可用的方法实现这个需求。

> PS：如果让你实现一个在二叉搜索树中通过排名计算对应元素的方法 `rank(int index)`，你会怎么设计？你可以思考一下，我会把答案写在留言区置顶。

除了平衡二叉树，还有没有什么常用的数据结构是动态有序的？优先级队列（二叉堆）行不行？

好像也不太行，因为优先级队列是一种受限的数据结构，只能从堆顶添加/删除元素，我们的 `addNum` 方法可以从堆顶插入元素，但是 `findMedian` 函数需要从数据中间取，这个功能优先级队列是没办法提供的。

可以看到，求个中位数还是挺难的，我们使尽浑身解数都没有一个高效地思路，下面直接来看解法吧，比较巧妙。

解法思路

我们必然需要有序数据结构，本题的核心思路是使用两个优先级队列。

——————————————

应合作方要求，本文不便在此发布，请扫码关注回复关键词「中位数」查看：

仗剑篇、进阶数据结构



进阶数据结构包括树和图，从定义上来说树是特殊的图，但实际场景中树和图的区别还是蛮大的，所以要分开说。

因为树算法大多涉及递归操作，而图有很多大家耳熟能详的经典算法，所以我把它们归为进阶数据结构。

公众号标签：[手把手刷数据结构](#)

2.1 二叉树

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

二叉树的重要性是我在公众号经常强调的，可以说 BFS 算法、DFS 算法、回溯算法、动态规划、分治算法、图论算法都是二叉树算法的衍生，其中都有二叉树题目的思维模式和代码框架。

我做的第一期专项训练营就是二叉树专题，足以看出我对这个专题的重视。

单单二叉树的遍历，如果往原理上问，就能考倒一大片人，比如你告诉我，二叉树的前序遍历结果怎么算?

你肯定可以写出如下代码：

```
List<Integer> res = new LinkedList<>();

// 返回前序遍历结果
List<Integer> preorder(TreeNode root) {
    traverse(root);
    return res;
}

// 二叉树遍历函数
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序遍历位置
    res.addLast(root.val);
    traverse(root.left);
    traverse(root.right);
}
```

还有别的想法吗？没有了？

还可以这样写：

```
// 定义：输入一棵二叉树的根节点，返回这棵树的前序遍历结果
List<Integer> preorder(TreeNode root) {
    List<Integer> res = new LinkedList<>();
    if (root == null) {
        return res;
    }
    // 前序遍历的结果，root.val 在第一个
    res.add(root.val);
    // 后面接着左子树的前序遍历结果
    res.addAll(preorder(root.left));
    // 最后接着右子树的前序遍历结果
```

```
    res.addAll(preorder(root.right));
}
```

这两个解法，前者是 回溯算法核心思路，后者是 动态规划核心思路，可以说你理解透彻二叉树，就成功了一半。

最后，我后续还会推出各个专题的训练营，持续关注我的公众号就好。

公众号标签：二叉树

东哥带你刷二叉树（第一期）

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

226. 翻转二叉树（简单）

114. 二叉树展开为链表（中等）

116. 填充每个节点的下一个右侧节点指针（中等）

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

我们公众号的成名之作 学习数据结构和算法的框架思维 中多次强调，先刷二叉树的题目，先刷二叉树的题目，先刷二叉树的题目，因为很多经典算法，以及我们前文讲过的所有回溯、动归、分治算法，其实都是树的问题，而树的问题就永远逃不开树的递归遍历框架这几行破代码：

```
/* 二叉树遍历框架 */
void traverse(TreeNode root) {
    // 前序遍历
    traverse(root.left);
    // 中序遍历
    traverse(root.right);
    // 后序遍历
}
```

上篇公众号文章让读者留言说说对什么问题还有疑惑，我接下来可以重点写一写相关的文章。结果还有很多读者说觉得「递归」非常难以理解，说实话，递归解法应该是最简单，最容易理解的才对，行云流水地写递归代码是学好算法的基本功，而二叉树相关的题目就是最练习递归基本功，最练习框架思维的。

我先花一些篇幅说明二叉树算法的重要性。

一、二叉树的重要性

举个例子，比如说我们的经典算法「快速排序」和「归并排序」，对于这两个算法，你有什么理解？**如果你告诉我，快速排序就是个二叉树的前序遍历，归并排序就是个二叉树的后序遍历，那么我就知道你是个算法高手了。**

为什么快速排序和归并排序能和二叉树扯上关系？我们来简单分析一下他们的算法思想和代码框架：

快速排序的逻辑是，若要对 nums[lo..hi] 进行排序，我们先找一个分界点 p，通过交换元素使得 nums[lo..p-1] 都小于等于 nums[p]，且 nums[p+1..hi] 都大于 nums[p]，然后递归地去 nums[lo..p-1] 和 nums[p+1..hi] 中寻找新的分界点，最后整个数组就被排序了。

快速排序的代码框架如下：

```
void sort(int[] nums, int lo, int hi) {
    /****** 前序遍历位置 ******/
    // 通过交换元素构建分界点 p
    int p = partition(nums, lo, hi);
    /************************/

    sort(nums, lo, p - 1);
    sort(nums, p + 1, hi);
}
```

先构造分界点，然后去左右子数组构造分界点，你看这不就是一个二叉树的前序遍历吗？

再说说归并排序的逻辑，若要对 nums[lo..hi] 进行排序，我们先对 nums[lo..mid] 排序，再对 nums[mid+1..hi] 排序，最后把这两个有序的子数组合并，整个数组就排好序了。

归并排序的代码框架如下：

```
// 定义：排序 nums[lo..hi]
void sort(int[] nums, int lo, int hi) {
    int mid = (lo + hi) / 2;
    // 排序 nums[lo..mid]
    sort(nums, lo, mid);
    // 排序 nums[mid+1..hi]
    sort(nums, mid + 1, hi);

    /****** 后序位置 ******/
    // 合并 nums[lo..mid] 和 nums[mid+1..hi]
    merge(nums, lo, mid, hi);
    /*********************/
}
```

先对左右子数组排序，然后合并（类似合并有序链表的逻辑），你看这是不是二叉树的后序遍历框架？另外，这不就是传说中的分治算法嘛，不过如此呀。

如果你一眼就识破这些排序算法的底细，还需要背这些算法代码吗？这不是手到擒来，从框架慢慢扩展就能写出算法了。

说了这么多，旨在说明，二叉树的算法思想的运用广泛，甚至可以说，只要涉及递归，都可以抽象成二叉树的问题，**所以本文和后续的 手把手带你刷二叉树（第二期） 以及 手把手刷二叉树（第三期），我们直接上几道比较有意思，且能体现出递归算法精妙的二叉树题目，东哥手把手教你怎么用算法框架搞定它们。**

二、写递归算法的秘诀

我们前文 二叉树的最近公共祖先 写过，**写递归算法的关键是要明确函数的「定义」是什么，然后相信这个定义，利用这个定义推导最终结果，绝不要跳入递归的细节。**

怎么理解呢，我们用一个具体的例子来说，比如说让你计算一棵二叉树共有几个节点：

```
// 定义：count(root) 返回以 root 为根的树有多少节点
int count(TreeNode root) {
    // base case
    if (root == null) return 0;
    // 自己加上子树的节点数就是整棵树的节点数
    return 1 + count(root.left) + count(root.right);
}
```

这个问题非常简单，大家应该都会写这段代码，`root` 本身就是一个节点，加上左右子树的节点数就是以 `root` 为根的树的节点总数。

左右子树的节点数怎么算？其实就是计算根为 `root.left` 和 `root.right` 两棵树的节点数呗，按照定义，递归调用 `count` 函数即可算出来。

写树相关的算法，简单说就是，先搞清楚当前 `root` 节点「该做什么」以及「什么时候做」，然后根据函数定义递归调用子节点，递归调用会让孩子节点做相同的事情。

所谓「该做什么」就是让你想清楚写什么代码能够实现题目想要的效果，所谓「什么时候做」，就是让你思考这段代码到底应该写在前序、中序还是后序遍历的代码位置上。

我们接下来看几道算法题目实操一下。

三、算法实践

第一题、翻转二叉树

我们先从简单的题开始，看看力扣第 226 题「翻转二叉树」，输入一个二叉树根节点 `root`，让你把整棵树镜像翻转，比如输入的二叉树如下：

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

算法原地翻转二叉树，使得以 `root` 为根的树变成：

```
     4
   /   \
  7     2
```

```
  / \   / \
 9   6 3   1
```

通过观察，**我们发现只要把二叉树上的每一个节点的左右子节点进行交换，最后的结果就是完全翻转之后的二叉树。**

可以直接写出解法代码：

```java
// 将整棵树的节点翻转
TreeNode invertTree(TreeNode root) {
    // base case
    if (root == null) {
        return null;
    }

    /**** 前序遍历位置 ****/
    // root 节点需要交换它的左右子节点
    TreeNode tmp = root.left;
    root.left = root.right;
    root.right = tmp;

    // 让左右子节点继续翻转它们的子节点
    invertTree(root.left);
    invertTree(root.right);

    return root;
}
```

这道题目比较简单，关键思路在于我们发现翻转整棵树就是交换每个节点的左右子节点，于是我们把交换左右子节点的代码放在了前序遍历的位置。

值得一提的是，如果把交换左右子节点的代码复制粘贴到后序遍历的位置也是可以的，但是直接放到中序遍历的位置是不行的，请你想一想为什么？这个应该不难想到，我会把答案置顶在公众号留言区。

首先讲这道题目是想告诉你，**二叉树题目的一个难点就是，如何把题目的要求细化成每个节点需要做的事情。**

这种洞察力需要多刷题训练，我们看下一道题。

第二题、填充二叉树节点的右侧指针

这是力扣第 116 题，看下题目：

116. 填充每个节点的下一个右侧节点指针 labuladong 题解 思路

难度 中等 239

给定一个**完美二叉树**，其所有叶子节点都在同一层，每个父节点都有两个子节点。二叉树定义如下：

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 `NULL`。

初始状态下，所有 next 指针都被设置为 `NULL`。

```
Node connect(Node root);
```

题目的意思就是把二叉树的每一层节点都用 `next` 指针连接起来：

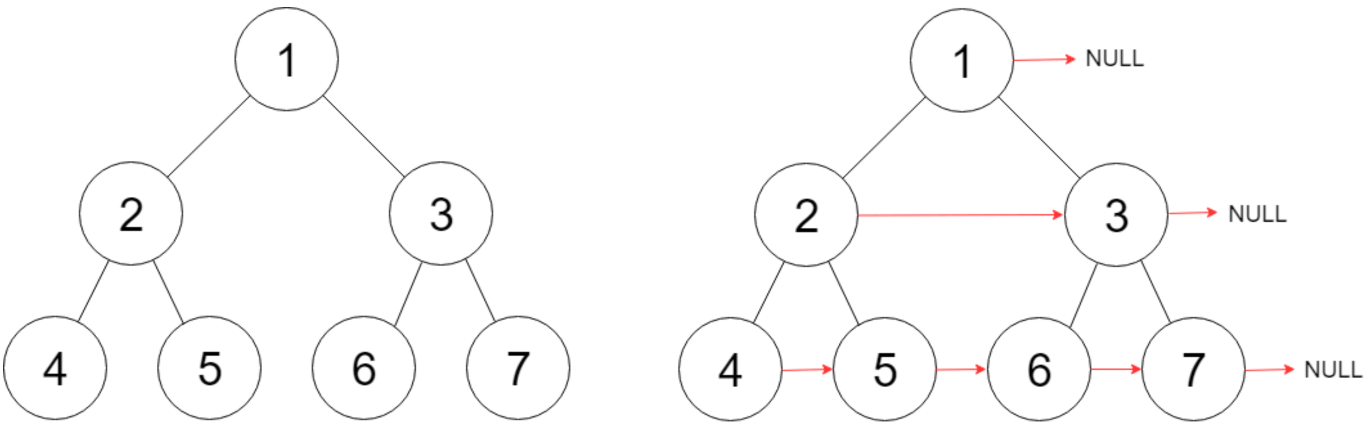


Figure A Figure B

而且题目说了，输入是一棵「完美二叉树」，形象地说整棵二叉树是一个正三角形，除了最右侧的节点 `next` 指针会指向 `null`，其他节点的右侧一定有相邻的节点。

这道题怎么做呢？把每一层的节点穿起来，是不是只要把每个节点的左右子节点都穿起来就行了？

我们可以模仿上一道题，写出如下代码：

```java
Node connect(Node root) {
    if (root == null || root.left == null) {
        return root;
    }

    root.left.next = root.right;

    connect(root.left);
    connect(root.right);

    return root;
}
```

这样其实有很大问题，再看看这张图：

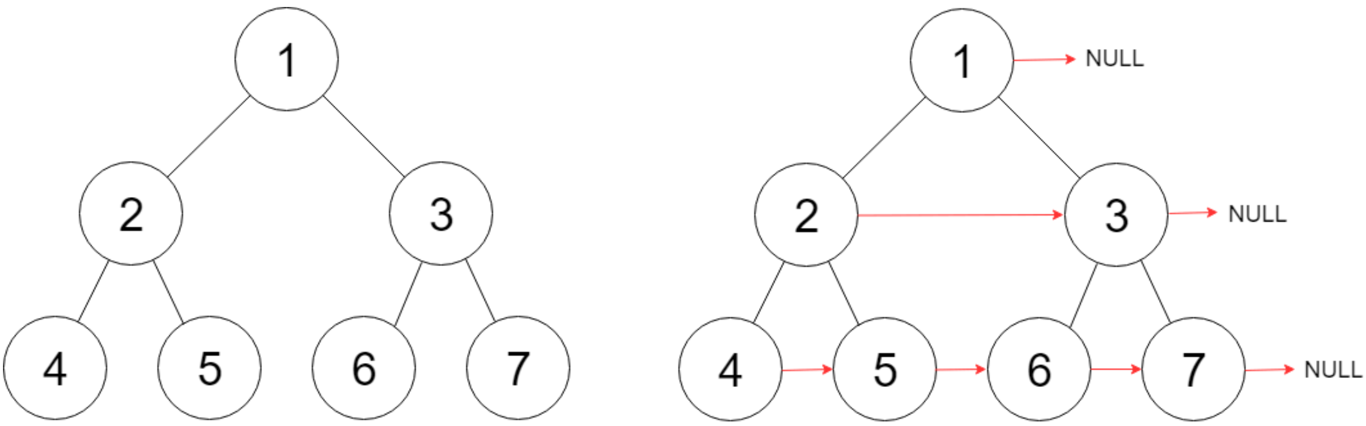


Figure A Figure B

节点 5 和节点 6 不属于同一个父节点，那么按照这段代码的逻辑，它俩就没办法被穿起来，这是不符合题意的。

回想刚才说的，**二叉树的问题难点在于，如何把题目的要求细化成每个节点需要做的事情**，但是如果只依赖一个节点的话，肯定是没办法连接「跨父节点」的两个相邻节点的。

那么，我们的做法就是增加函数参数，一个节点做不到，我们就给他安排两个节点，「将每一层二叉树节点连接起来」可以细化成「将每两个相邻节点都连接起来」：

```java
// 主函数
Node connect(Node root) {
    if (root == null) return null;
    connectTwoNode(root.left, root.right);
    return root;
}

// 辅助函数
void connectTwoNode(Node node1, Node node2) {
    if (node1 == null || node2 == null) {
        return;
    }
    /**** 前序遍历位置 ****/
```

```
    // 将传入的两个节点连接
    node1.next = node2;

    // 连接相同父节点的两个子节点
    connectTwoNode(node1.left, node1.right);
    connectTwoNode(node2.left, node2.right);
    // 连接跨越父节点的两个子节点
    connectTwoNode(node1.right, node2.left);
}
```

这样，`connectTwoNode` 函数不断递归，可以无死角覆盖整棵二叉树，将所有相邻节点都连接起来，也就避免了我们之前出现的问题，这道题就解决了。

第三题、将二叉树展开为链表

这是力扣第 114 题，看下题目：

114. 二叉树展开为链表 labuladong 题解 思路

难度 中等 548

给定一个二叉树，原地将它展开为一个单链表。例如，给定二叉树：

```
    1
   / \
  2   5
 / \   \
3   4   6
```

将其展开为：

```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
```

函数签名如下：

```
void flatten(TreeNode root);
```

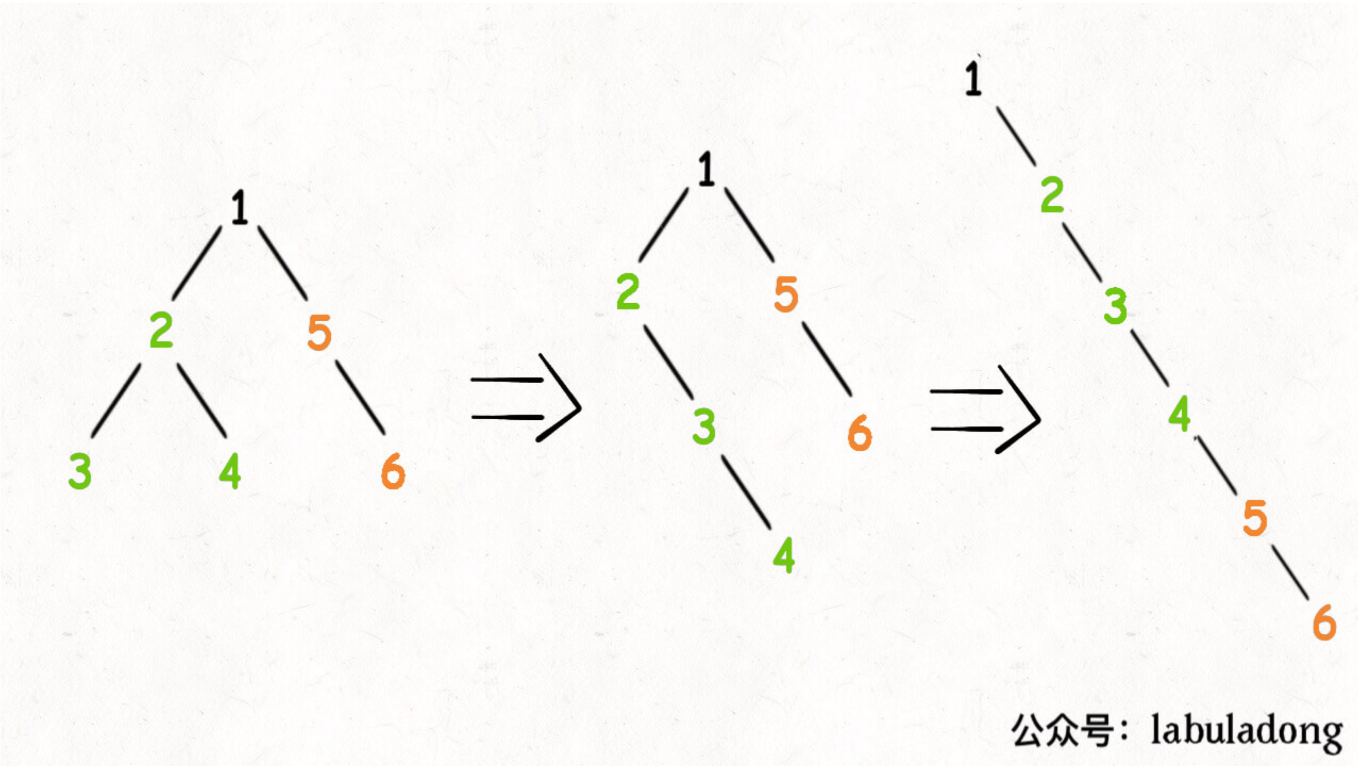
我们尝试给出这个函数的定义：

给 `flatten` 函数输入一个节点 `root`，那么以 `root` 为根的二叉树就会被拉平为一条链表。

我们再梳理一下，如何按题目要求把一棵树拉平成一条链表？很简单，以下流程：

1、将 `root` 的左子树和右子树拉平。

2、将 `root` 的右子树接到左子树下方，然后将整个左子树作为右子树。



上面三步看起来最难的应该是第一步对吧，如何把 `root` 的左右子树拉平？其实很简单，按照 `flatten` 函数的定义，对 `root` 的左右子树递归调用 `flatten` 函数即可：

```
// 定义：将以 root 为根的树拉平为链表
void flatten(TreeNode root) {
    // base case
    if (root == null) return;

    flatten(root.left);
    flatten(root.right);

    /**** 后序遍历位置 ****/
    // 1、左右子树已经被拉平成一条链表
    TreeNode left = root.left;
    TreeNode right = root.right;

    // 2、将左子树作为右子树
    root.left = null;
    root.right = left;

    // 3、将原先的右子树接到当前右子树的末端
    TreeNode p = root;
    while (p.right != null) {
        p = p.right;
```

```
        }
        p.right = right;
    }
```

你看，这就是递归的魅力，你说 `flatten` 函数是怎么把左右子树拉平的？说不清楚，但是只要知道 `flatten` 的定义如此，相信这个定义，让 `root` 做它该做的事情，然后 `flatten` 函数就会按照定义工作。另外注意递归框架是后序遍历，因为我们要先拉平左右子树才能进行后续操作。

至此，这道题也解决了，我们旧文 k个一组翻转链表 的递归思路和本题也有一些类似。

四、最后总结

递归算法的关键要明确函数的定义，相信这个定义，而不要跳进递归细节。

写二叉树的算法题，都是基于递归框架的，我们先要搞清楚 `root` 节点它自己要做什么，然后根据题目要求选择使用前序，中序，后续的递归框架。

二叉树题目的难点在于如何通过题目的要求思考出每一个节点需要做什么，这个只能通过多刷题进行练习了。

如果本文讲的三道题对你有一些启发，请三连，数据好的话东哥下次再来一波手把手刷题文，你会发现二叉树的题真的是越刷越顺手，欲罢不能，恨不得一口气把二叉树的题刷通。

接下来可阅读：

- 手把手刷二叉树（第二期）
- 手把手刷二叉树（第三期）

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



东哥带你刷二叉树（第二期）

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

654. 最大二叉树（中等）

105. 从前序与中序遍历序列构造二叉树（中等）

106. 从中序与后序遍历序列构造二叉树（中等）

889. 根据前序和后序遍历构造二叉树（中等）

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

上篇文章 手把手教你刷二叉树（第一篇） 连刷了三道二叉树题目，很多读者直呼内行。其实二叉树相关的算法真的不难，本文再来三道，手把手带你看看树的算法到底怎么做。

先来复习一下，我们说过写树的算法，关键思路如下：

把题目的要求细化，搞清楚根节点应该做什么，然后剩下的事情抛给前/中/后序的遍历框架就行了，我们千万不要跳进递归的细节里，你的脑袋才能压几个栈呀。

也许你还不太理解这句话，我们下面来看例子。

构造最大二叉树

先来道简单的，这是力扣第 654 题，题目如下：

654. 最大二叉树 labuladong 题解 思路

难度 中等 190

给定一个不含重复元素的整数数组。一个以此数组构建的最大二叉树定义如下：

1. 二叉树的根是数组中的最大元素。
2. 左子树是通过数组中最大值左边部分构造出的最大二叉树。
3. 右子树是通过数组中最大值右边部分构造出的最大二叉树。

通过给定的数组构建最大二叉树，并且输出这个树的根节点。

示例：

```
输入：[3,2,1,6,0,5]
输出：返回下面这棵树的根节点：

      6
    /   \
   3     5
    \    /
     2  0
       \
        1
```

函数签名如下：

```
TreeNode constructMaximumBinaryTree(int[] nums);
```

按照我们刚才说的，先明确根节点做什么？**对于构造二叉树的问题，根节点要做的就是把想办法把自己构造出来。**

我们肯定要遍历数组把找到最大值 `maxVal`，把根节点 `root` 做出来，然后对 `maxVal` 左边的数组和右边的数组进行递归调用，作为 `root` 的左右子树。

按照题目给出的例子，输入的数组为 `[3,2,1,6,0,5]`，对于整棵树的根节点来说，其实在做这件事：

```
TreeNode constructMaximumBinaryTree([3,2,1,6,0,5]) {
    // 找到数组中的最大值
    TreeNode root = new TreeNode(6);
    // 递归调用构造左右子树
    root.left = constructMaximumBinaryTree([3,2,1]);
    root.right = constructMaximumBinaryTree([0,5]);
    return root;
}
```

再详细一点，就是如下伪码：

```
TreeNode constructMaximumBinaryTree(int[] nums) {
    if (nums is empty) return null;
    // 找到数组中的最大值
    int maxVal = Integer.MIN_VALUE;
    int index = 0;
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] > maxVal) {
            maxVal = nums[i];
            index = i;
        }
    }

    TreeNode root = new TreeNode(maxVal);
    // 递归调用构造左右子树
    root.left = constructMaximumBinaryTree(nums[0..index-1]);
    root.right = constructMaximumBinaryTree(nums[index+1..nums.length-1]);
    return root;
}
```

看懂了吗？**对于每个根节点，只需要找到当前 `nums` 中的最大值和对应的索引，然后递归调用左右数组构造左右子树即可**。

明确了思路，我们可以重新写一个辅助函数 `build`，来控制 `nums` 的索引：

```
/* 主函数 */
TreeNode constructMaximumBinaryTree(int[] nums) {
    return build(nums, 0, nums.length - 1);
}

/* 将 nums[lo..hi] 构造成符合条件的树，返回根节点 */
TreeNode build(int[] nums, int lo, int hi) {
    // base case
    if (lo > hi) {
        return null;
    }

    // 找到数组中的最大值和对应的索引
```

```
    int index = -1, maxVal = Integer.MIN_VALUE;
    for (int i = lo; i <= hi; i++) {
        if (maxVal < nums[i]) {
            index = i;
            maxVal = nums[i];
        }
    }

    TreeNode root = new TreeNode(maxVal);
    // 递归调用构造左右子树
    root.left = build(nums, lo, index - 1);
    root.right = build(nums, index + 1, hi);

    return root;
}
```

至此，这道题就做完了，还是挺简单的对吧，下面看两道更困难一些的。

通过前序和中序遍历结果构造二叉树

经典问题了，面试/笔试中常考，力扣第 105 题就是这个问题：

105. 从前序与中序遍历序列构造二叉树

难度 中等 679

根据一棵树的前序遍历与中序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

例如，给出

```
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]
```

返回如下的二叉树:

```
    3
   / \
  9  20
    /  \
   15   7
```

函数签名如下：

```
TreeNode buildTree(int[] preorder, int[] inorder);
```

废话不多说，直接来想思路，首先思考，根节点应该做什么。

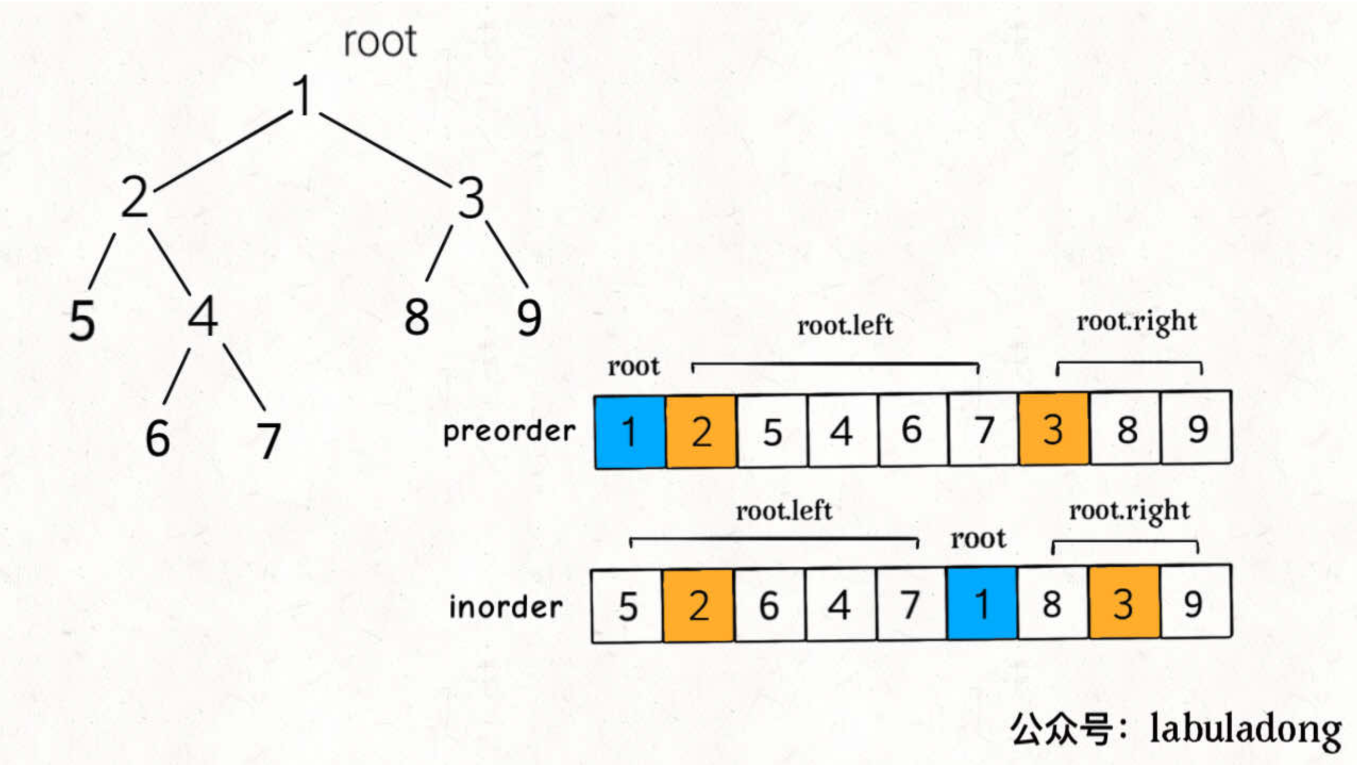
类似上一题，我们肯定要想办法确定根节点的值，把根节点做出来，然后递归构造左右子树即可。

我们先来回顾一下，前序遍历和中序遍历的结果有什么特点？

```
void traverse(TreeNode root) {
    // 前序遍历
    preorder.add(root.val);
    traverse(root.left);
    traverse(root.right);
}

void traverse(TreeNode root) {
    traverse(root.left);
    // 中序遍历
    inorder.add(root.val);
    traverse(root.right);
}
```

前文 二叉树就那几个框架 写过，这样的遍历顺序差异，导致了 `preorder` 和 `inorder` 数组中的元素分布有如下特点：



找到根节点是很简单的，前序遍历的第一个值 `preorder[0]` 就是根节点的值，关键在于如何通过根节点的值，将 `preorder` 和 `postorder` 数组划分成两半，构造根节点的左右子树？

换句话说，对于以下代码中的 `?` 部分应该填入什么：

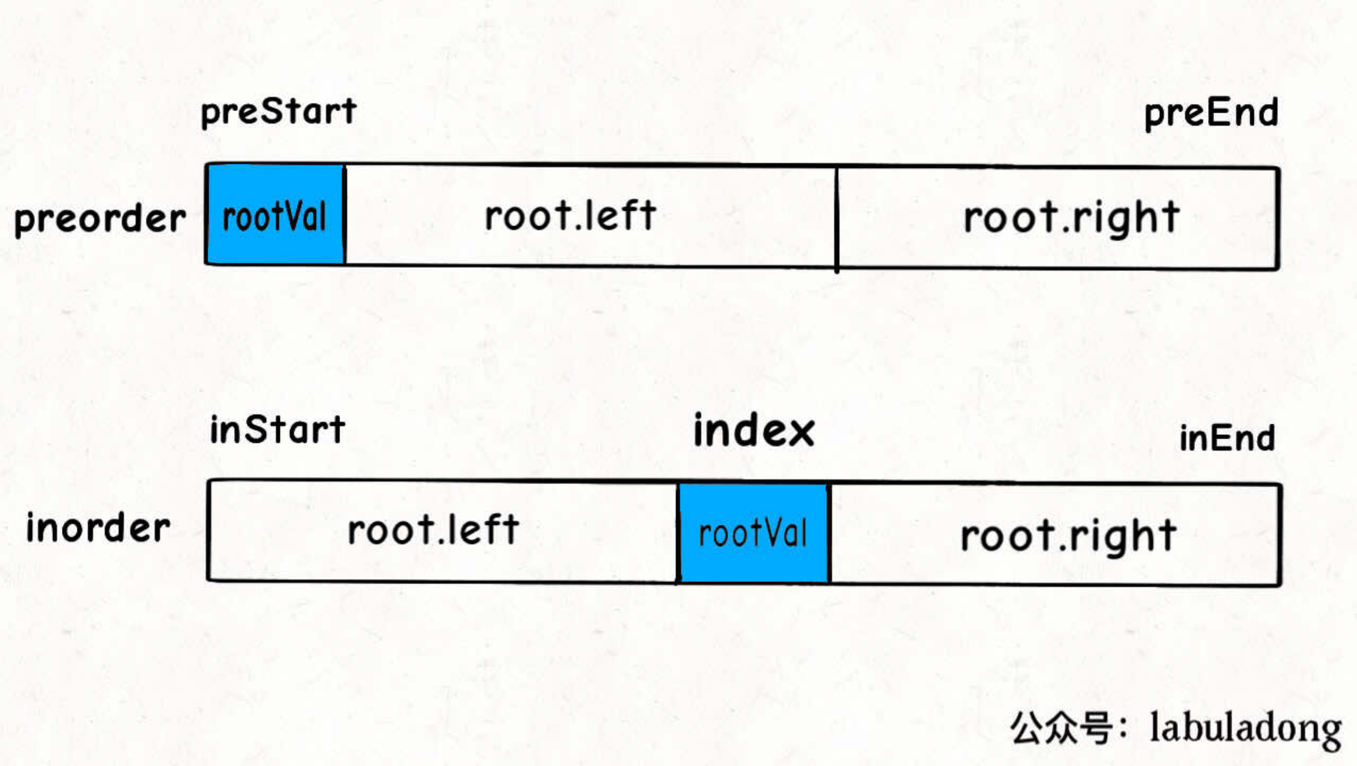
```java
/* 主函数 */
TreeNode buildTree(int[] preorder, int[] inorder) {
    return build(preorder, 0, preorder.length - 1,
                 inorder, 0, inorder.length - 1);
}

/*
   若前序遍历数组为 preorder[preStart..preEnd],
   中序遍历数组为 inorder[inStart..inEnd],
   构造二叉树，返回该二叉树的根节点
*/
TreeNode build(int[] preorder, int preStart, int preEnd,
               int[] inorder, int inStart, int inEnd) {
    // root 节点对应的值就是前序遍历数组的第一个元素
    int rootVal = preorder[preStart];
    // rootVal 在中序遍历数组中的索引
    int index = 0;
    for (int i = inStart; i <= inEnd; i++) {
        if (inorder[i] == rootVal) {
            index = i;
            break;
        }
    }

    TreeNode root = new TreeNode(rootVal);
    // 递归构造左右子树
    root.left = build(preorder, ?, ?,
                      inorder, ?, ?);

    root.right = build(preorder, ?, ?,
                       inorder, ?, ?);
    return root;
}
```

对于代码中的 `rootVal` 和 `index` 变量，就是下图这种情况：

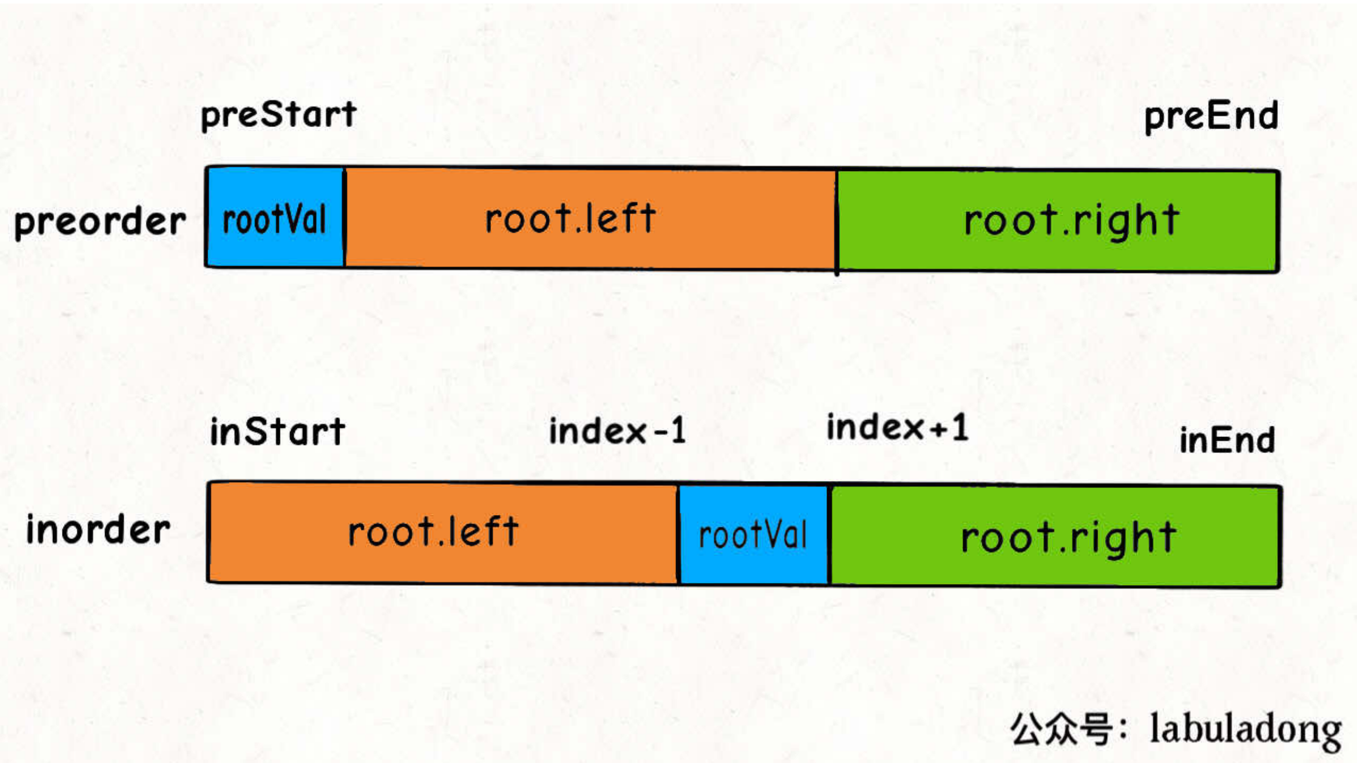


现在我们来看图做填空题，下面这几个问号处应该填什么：

```
root.left = build(preorder, ?, ?,
                  inorder, ?, ?);

root.right = build(preorder, ?, ?,
                   inorder, ?, ?);
```

对于左右子树对应的 `inorder` 数组的起始索引和终止索引比较容易确定：

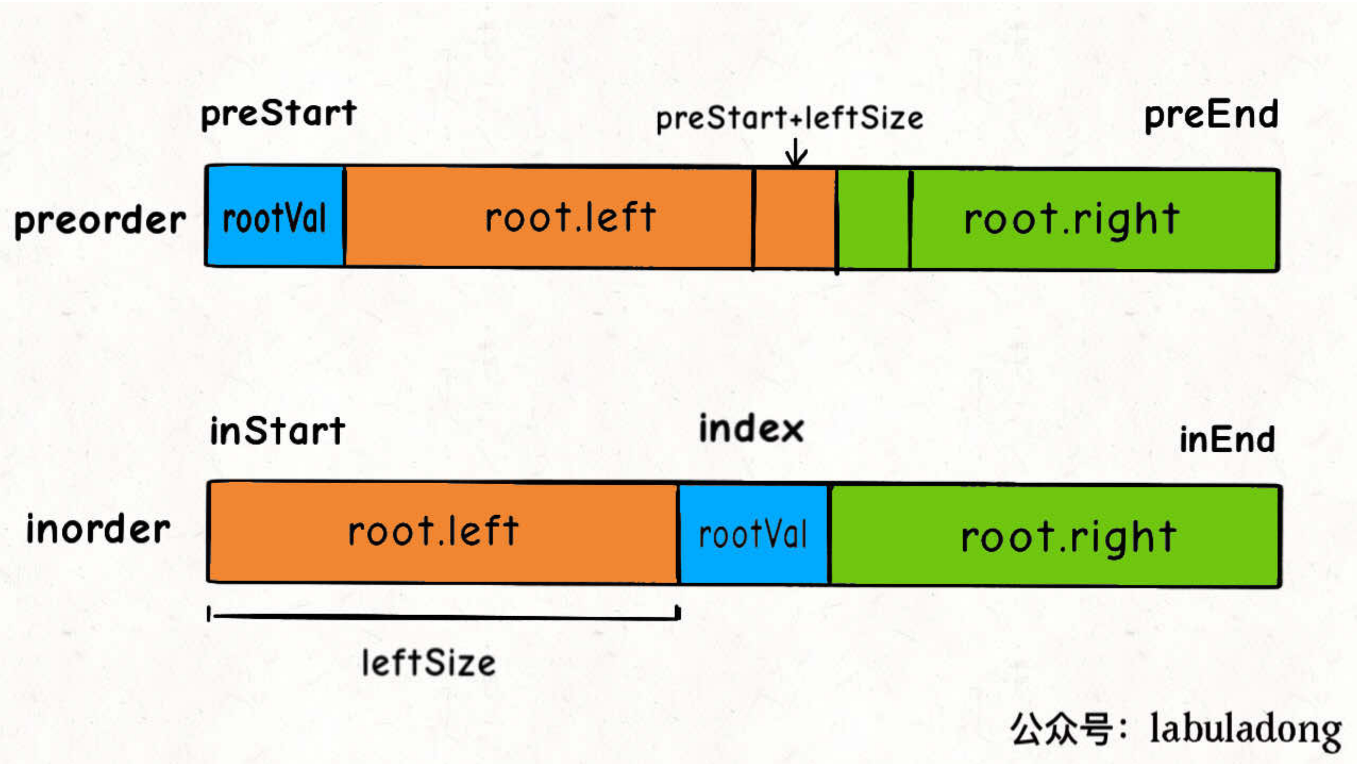


```
root.left = build(preorder, ?, ?,
                  inorder, inStart, index - 1);

root.right = build(preorder, ?, ?,
                   inorder, index + 1, inEnd);
```

对于 `preorder` 数组呢？如何确定左右数组对应的起始索引和终止索引？

这个可以通过左子树的节点数推导出来，假设左子树的节点数为 `leftSize`，那么 `preorder` 数组上的索引情况是这样的：



看着这个图就可以把 `preorder` 对应的索引写进去了：

```
int leftSize = index - inStart;

root.left = build(preorder, preStart + 1, preStart + leftSize,
                  inorder, inStart, index - 1);

root.right = build(preorder, preStart + leftSize + 1, preEnd,
                   inorder, index + 1, inEnd);
```

至此，整个算法思路就完成了，我们再补一补 base case 即可写出解法代码：

```
TreeNode build(int[] preorder, int preStart, int preEnd,
               int[] inorder, int inStart, int inEnd) {

    if (preStart > preEnd) {
```

```java
            return null;
        }

        // root 节点对应的值就是前序遍历数组的第一个元素
        int rootVal = preorder[preStart];
        // rootVal 在中序遍历数组中的索引
        int index = 0;
        for (int i = inStart; i <= inEnd; i++) {
            if (inorder[i] == rootVal) {
                index = i;
                break;
            }
        }

        int leftSize = index - inStart;

        // 先构造出当前根节点
        TreeNode root = new TreeNode(rootVal);
        // 递归构造左右子树
        root.left = build(preorder, preStart + 1, preStart + leftSize,
                          inorder, inStart, index - 1);

        root.right = build(preorder, preStart + leftSize + 1, preEnd,
                           inorder, index + 1, inEnd);
        return root;
    }
```

我们的主函数只要调用 `build` 函数即可，你看着函数这么多参数，解法这么多代码，似乎比我们上面讲的那道题难很多，让人望而生畏，实际上呢，这些参数无非就是控制数组起止位置的，画个图就能解决了。

通过后序和中序遍历结果构造二叉树

类似上一题，这次我们利用**后序**和**中序**遍历的结果数组来还原二叉树，这是力扣第 106 题：

106. 从中序与后序遍历序列构造二叉树

难度 中等 296

根据一棵树的中序遍历与后序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

例如，给出

```
中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]
```

返回如下的二叉树:

```
    3
   / \
  9  20
    /  \
   15   7
```

函数签名如下：

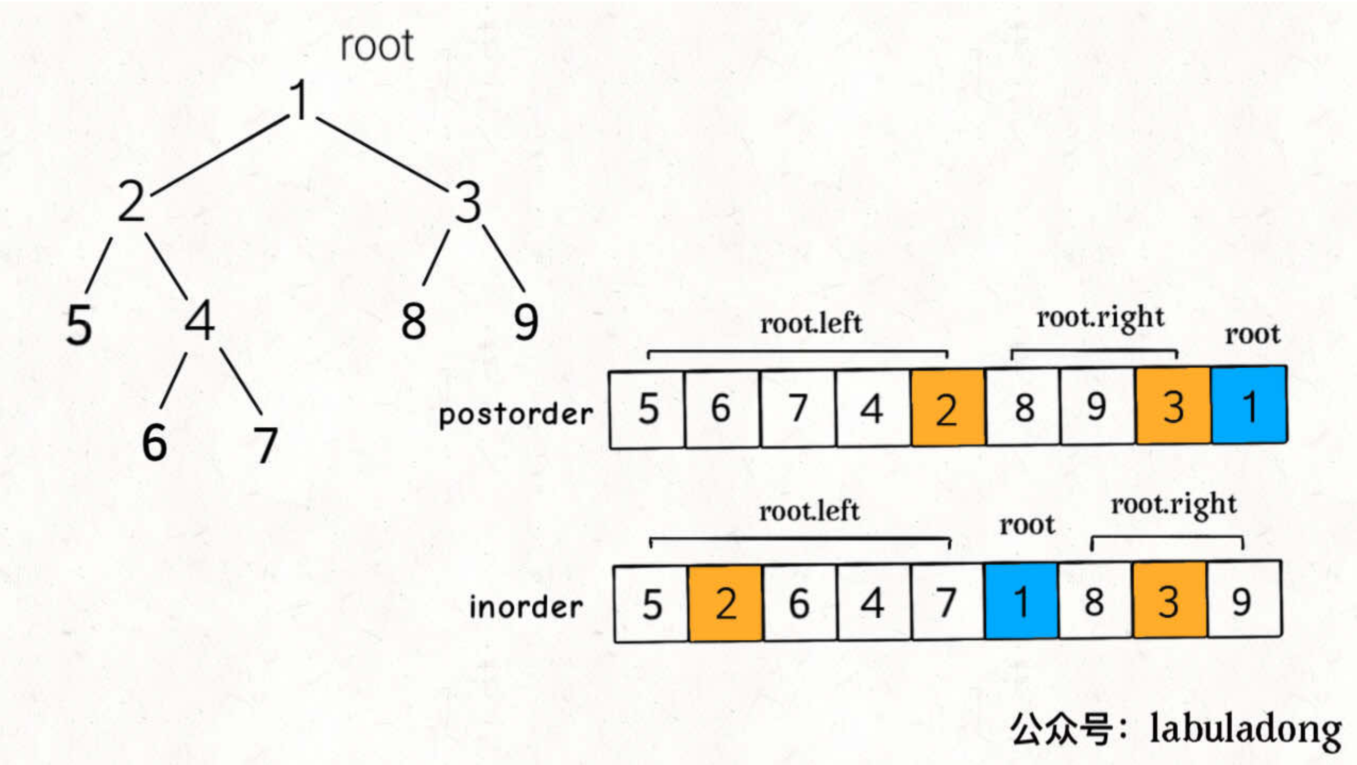
```
TreeNode buildTree(int[] inorder, int[] postorder);
```

类似的，看下后序和中序遍历的特点：

```
void traverse(TreeNode root) {
    traverse(root.left);
    traverse(root.right);
    // 后序遍历
    postorder.add(root.val);
}

void traverse(TreeNode root) {
    traverse(root.left);
    // 中序遍历
    inorder.add(root.val);
    traverse(root.right);
}
```

这样的遍历顺序差异，导致了 `preorder` 和 `inorder` 数组中的元素分布有如下特点：



这道题和上一题的关键区别是，后序遍历和前序遍历相反，根节点对应的值为 `postorder` 的最后一个元素。

整体的算法框架和上一题非常类似，我们依然写一个辅助函数 `build`：

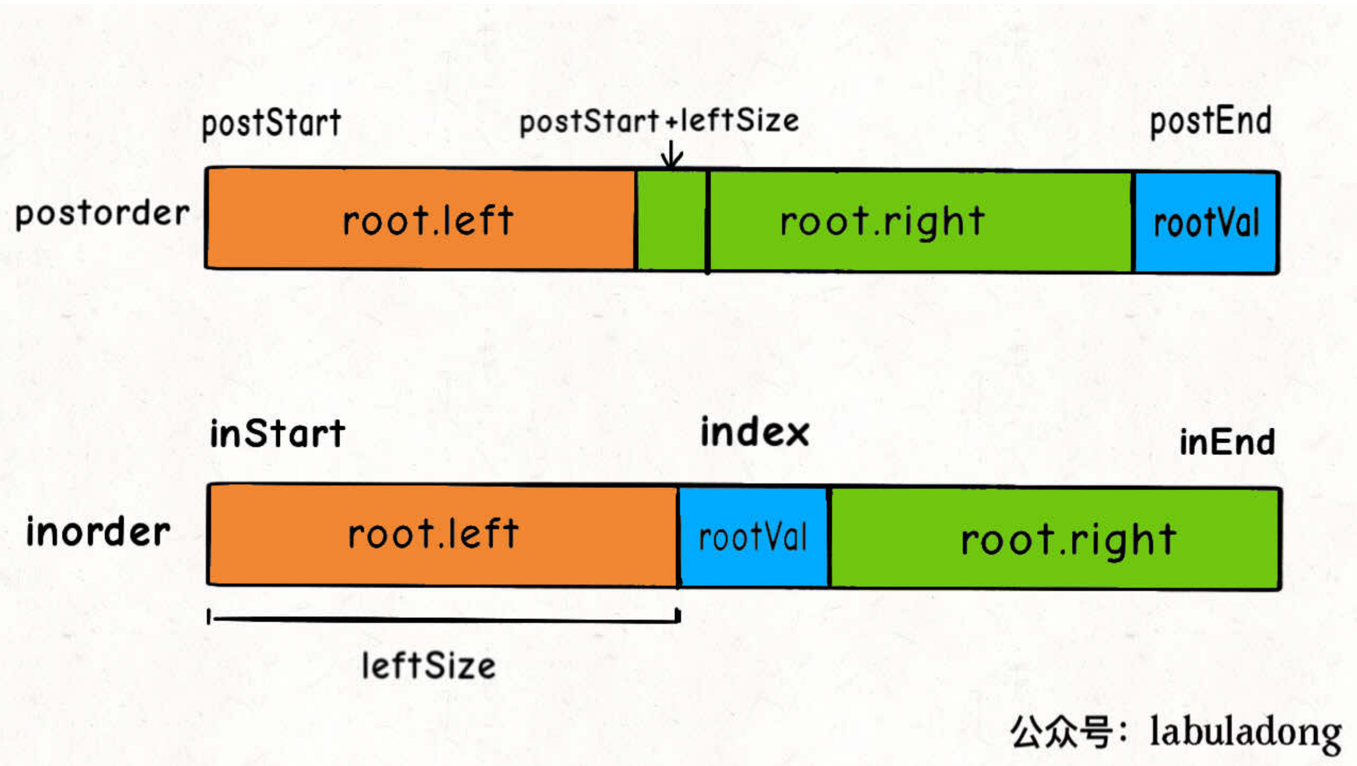
```
TreeNode buildTree(int[] inorder, int[] postorder) {
    return build(inorder, 0, inorder.length - 1,
                 postorder, 0, postorder.length - 1);
}
```

```java
/*
   后序遍历数组为 postorder[postStart..postEnd],
   中序遍历数组为 inorder[inStart..inEnd],
   构造二叉树，返回该二叉树的根节点
*/
TreeNode build(int[] inorder, int inStart, int inEnd,
               int[] postorder, int postStart, int postEnd) {
    // root 节点对应的值就是后序遍历数组的最后一个元素
    int rootVal = postorder[postEnd];
    // rootVal 在中序遍历数组中的索引
    int index = 0;
    for (int i = inStart; i <= inEnd; i++) {
        if (inorder[i] == rootVal) {
            index = i;
            break;
        }
    }

    TreeNode root = new TreeNode(rootVal);
    // 递归构造左右子树
    root.left = build(preorder, ?, ?,
                      inorder, ?, ?);

    root.right = build(preorder, ?, ?,
                       inorder, ?, ?);
    return root;
}
```

现在 `postoder` 和 `inorder` 对应的状态如下：



我们可以按照上图将问号处的索引正确填入：

```
int leftSize = index - inStart;

root.left = build(inorder, inStart, index - 1,
                  postorder, postStart, postStart + leftSize - 1);

root.right = build(inorder, index + 1, inEnd,
                   postorder, postStart + leftSize, postEnd - 1);
```

综上，可以写出完整的解法代码：

```
TreeNode build(int[] inorder, int inStart, int inEnd,
               int[] postorder, int postStart, int postEnd) {

    if (inStart > inEnd) {
        return null;
    }
    // root 节点对应的值就是后序遍历数组的最后一个元素
    int rootVal = postorder[postEnd];
    // rootVal 在中序遍历数组中的索引
    int index = 0;
    for (int i = inStart; i <= inEnd; i++) {
        if (inorder[i] == rootVal) {
            index = i;
            break;
        }
    }
    // 左子树的节点个数
    int leftSize = index - inStart;
    TreeNode root = new TreeNode(rootVal);
    // 递归构造左右子树
    root.left = build(inorder, inStart, index - 1,
                      postorder, postStart, postStart + leftSize - 1);

    root.right = build(inorder, index + 1, inEnd,
                       postorder, postStart + leftSize, postEnd - 1);
    return root;
}
```

有了前一题的铺垫，这道题很快就解决了，无非就是 `rootVal` 变成了最后一个元素，再改改递归函数的参数而已，只要明白二叉树的特性，也不难写出来。

通过后序和前序遍历结果构造二叉树

这是力扣第 889 题「根据前序和后序遍历构造二叉树」，给你输入二叉树的前序和后序遍历结果，让你还原二叉树的结构。

函数签名如下：

```java
TreeNode constructFromPrePost(int[] preorder, int[] postorder);
```

这道题和前两道题有一个本质的区别：

通过前序中序，或者后序中序遍历结果可以确定一棵原始二叉树，但是通过前序后序遍历结果无法确定原始二叉树。

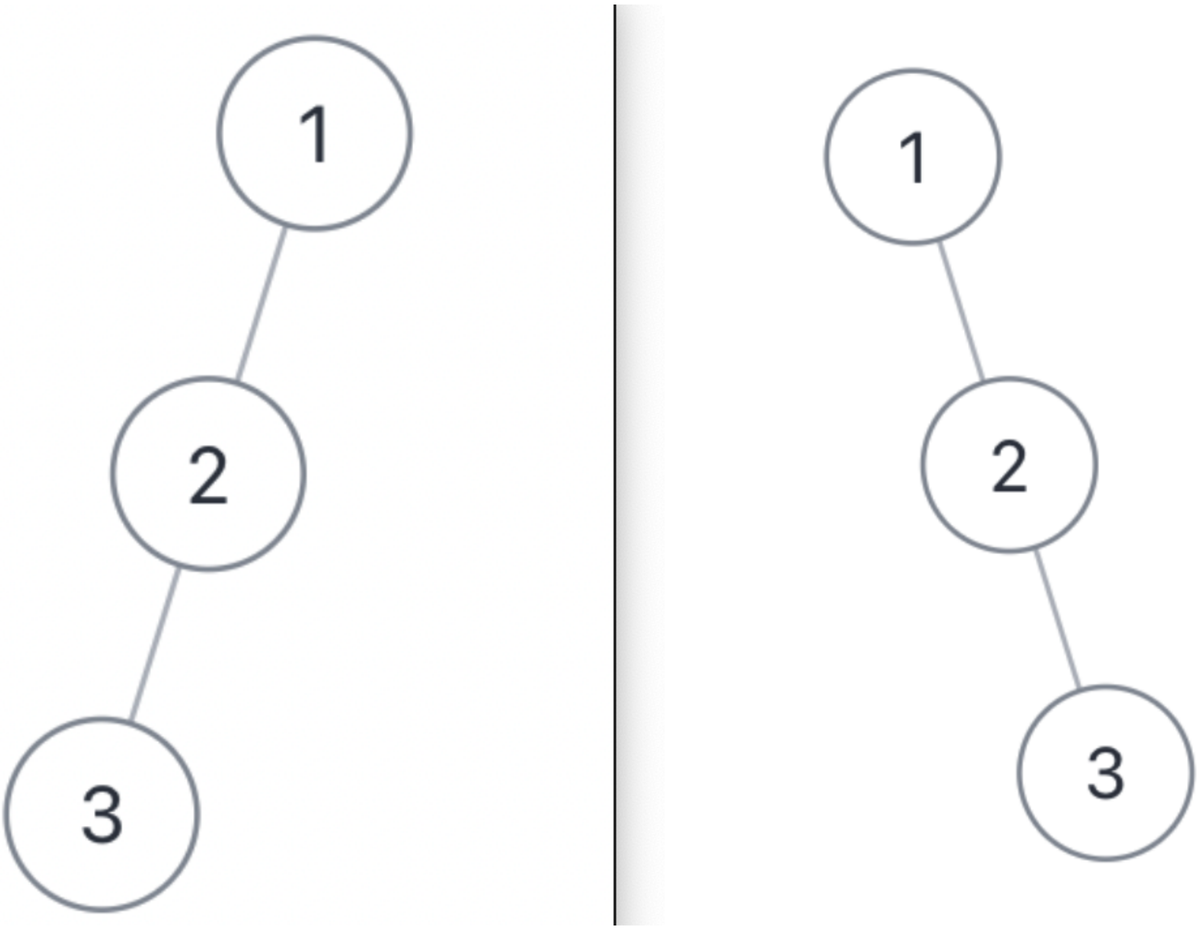
题目也说了，如果有多种可能的还原结果，你可以返回任意一种。

为什么呢？我们说过，构建二叉树的套路很简单，先找到根节点，然后找到并递归构造左右子树即可。

前两道题，可以通过前序或者后序遍历结果找到根节点，然后根据中序遍历结果确定左右子树。

这道题，你可以确定根节点，但是无法确切的知道左右子树有哪些节点。

举个例子，下面这两棵树结构不同，但是它们的前序遍历和后序遍历结果是相同的：

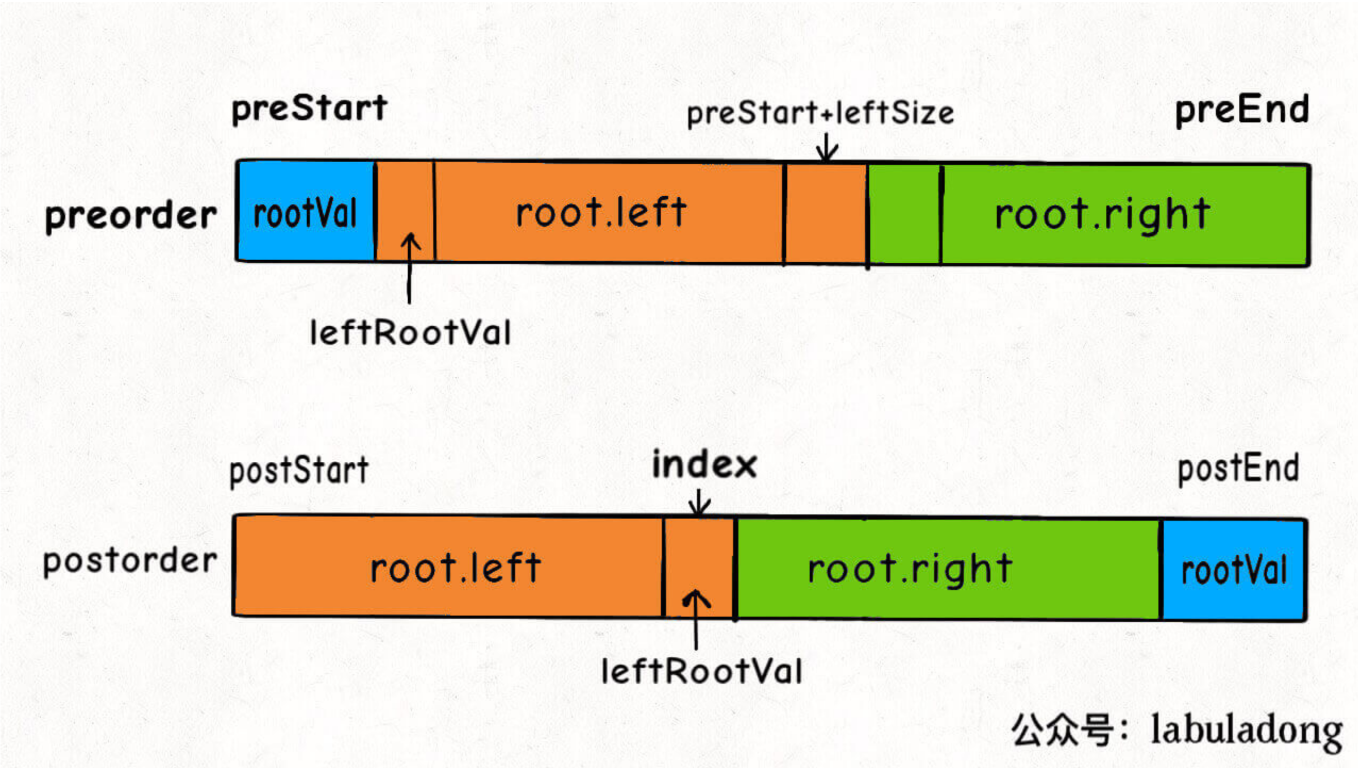


不过话说回来，用后序遍历和前序遍历结果还原二叉树，解法逻辑上和前两道题差别不大，也是通过控制左右子树的索引来构建：

1、首先把前序遍历结果的第一个元素或者后序遍历结果的最后一个元素确定为根节点的值。

2、然后把前序遍历结果的第二个元素作为左子树的根节点的值。

3、在后序遍历结果中寻找左子树根节点的值，从而确定了左子树的索引边界，进而确定右子树的索引边界，递归构造左右子树即可。



详情见代码。

```
class Solution {
    public TreeNode constructFromPrePost(int[] preorder, int[] postorder)
{
        return build(preorder, 0, preorder.length - 1,
                postorder, 0, postorder.length - 1);
    }

    // 定义：根据 preorder[preStart..preEnd] 和 postorder[postStart..postEnd]
    // 构建二叉树，并返回根节点。
    TreeNode build(int[] preorder, int preStart, int preEnd,
                   int[] postorder, int postStart, int postEnd) {
        if (preStart > preEnd) {
            return null;
        }
        if (preStart == preEnd) {
            return new TreeNode(preorder[preStart]);
        }

        // root 节点对应的值就是前序遍历数组的第一个元素
        int rootVal = preorder[preStart];
        // root.left 的值是前序遍历第二个元素
        // 通过前序和后序遍历构造二叉树的关键在于通过左子树的根节点
        // 确定 preorder 和 postorder 中左右子树的元素区间
        int leftRootVal = preorder[preStart + 1];
        // leftRootVal 在后序遍历数组中的索引
        int index = 0;
        for (int i = postStart; i < postEnd; i++) {
            if (postorder[i] == leftRootVal) {
                index = i;
```

```java
                    break;
                }
            }
            // 左子树的元素个数
            int leftSize = index - postStart + 1;

            // 先构造出当前根节点
            TreeNode root = new TreeNode(rootVal);
            // 递归构造左右子树
            // 根据左子树的根节点索引和元素个数推导左右子树的索引边界
            root.left = build(preorder, preStart + 1, preStart + leftSize,
                    postorder, postStart, index);
            root.right = build(preorder, preStart + leftSize + 1, preEnd,
                    postorder, index + 1, postEnd - 1);

            return root;
        }
    }
```

代码和前两道题非常类似，我们可以看着代码思考一下，为什么通过前序遍历和后序遍历结果还原的二叉树可能不唯一呢？

关键在这一句：

```java
int leftRootVal = preorder[preStart + 1];
```

我们假设前序遍历的第二个元素是左子树的根节点，但实际上左子树可能是空指针，这个元素可能是右子树的根节点。

由于这里无法确切进行判断，所以导致了最终答案的不唯一。

至此，通过前序和后序遍历结果还原二叉树的问题也解决了。

最后呼应下前文，**做二叉树的问题，关键是把题目的要求细化，搞清楚根节点应该做什么，然后剩下的事情抛给前/中/后序的遍历框架就行了**。

现在你是否明白其中的玄妙了呢？

接下来可阅读：

- 手把手刷二叉树（第三期）

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

东哥带你刷二叉树（第三期）

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

652. 寻找重复的子树（中等）

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

接前文 手把手带你刷二叉树（第一期） 和 手把手带你刷二叉树（第二期），本文继续来刷二叉树。

从前两篇文章的阅读量来看，大家还是能够通过二叉树学习到 框架思维 的。但还是有不少读者有一些问题，比如**如何判断我们应该用前序还是中序还是后序遍历的框架**？

那么本文就针对这个问题，不贪多，给你掰开揉碎只讲一道题。还是那句话，**根据题意，思考一个二叉树节点需要做什么，到底用什么遍历顺序就清楚了。**

看题，这是力扣第 652 题「寻找重复的子树」：

652. 寻找重复的子树 labuladong 题解 思路

难度 中等 162

给定一棵二叉树，返回所有重复的子树。对于同一类的重复子树，你只需要返回其中任意**一棵**的根结点即可。

两棵树重复是指它们具有相同的结构以及相同的结点值。

示例 1：

```
        1
       / \
      2   3
     /   / \
    4   2   4
       /
      4
```

下面是两个重复的子树：

```
      2
     /
    4
```

和

```
    4
```

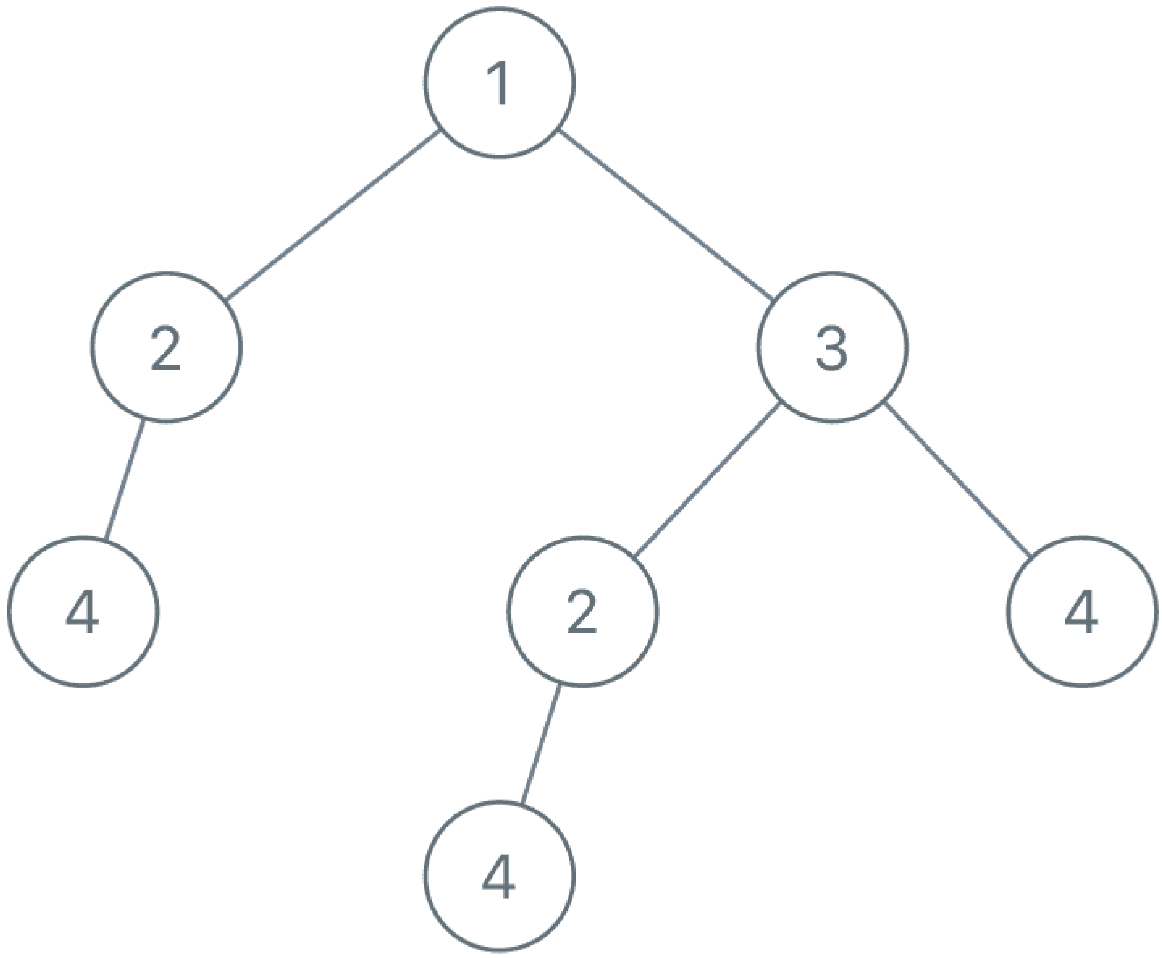
因此，你需要以列表的形式返回上述重复子树的根结点。

函数签名如下：

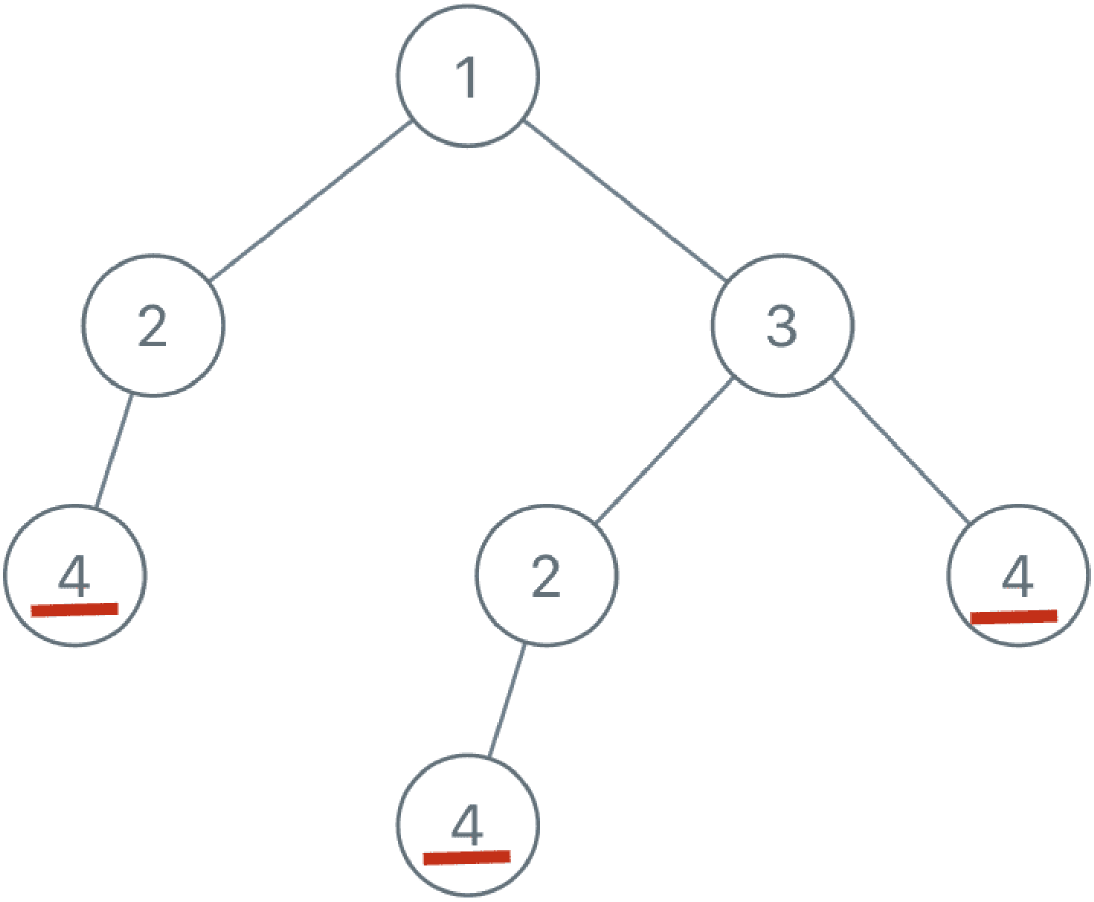
```
List<TreeNode> findDuplicateSubtrees(TreeNode root);
```

我来简单解释下题目，输入是一棵二叉树的根节点 `root`，返回的是一个列表，里面装着若干个二叉树节点，这些节点对应的子树在原二叉树中是存在重复的。

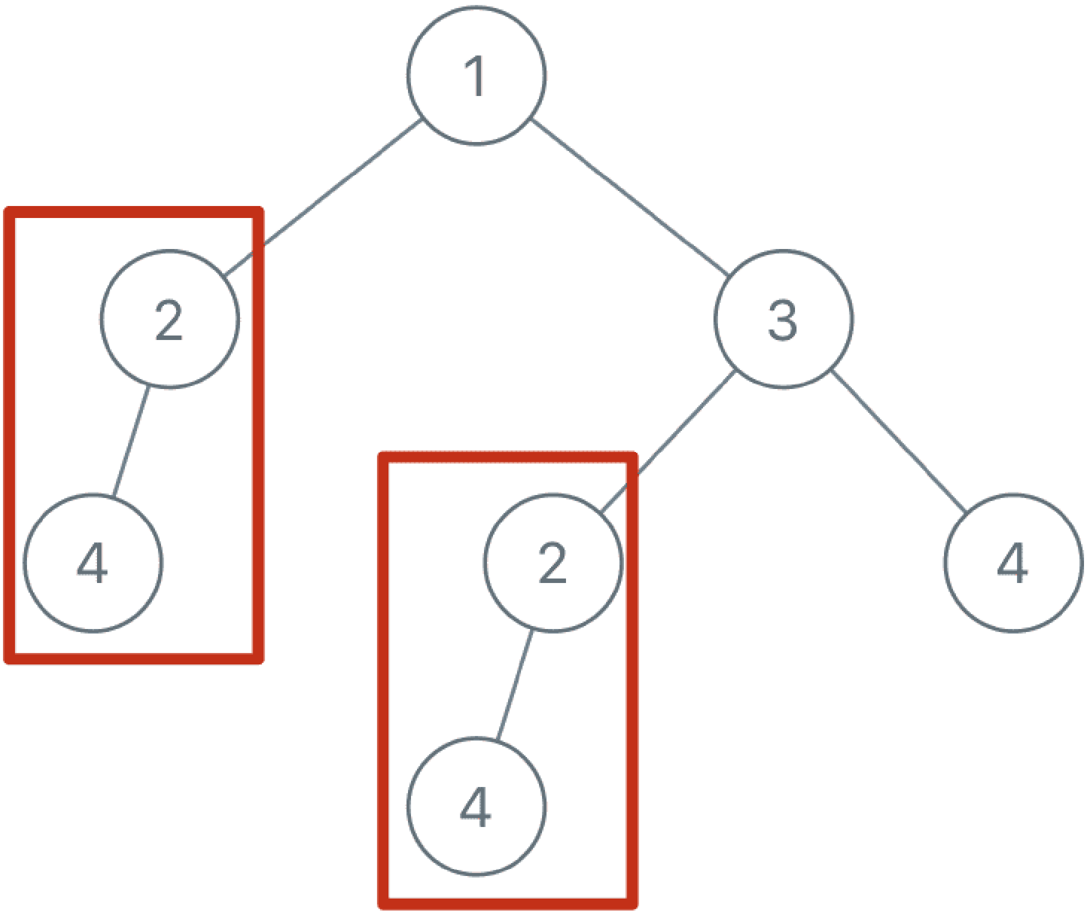
说起来比较绕，举例来说，比如输入如下的二叉树：



首先，节点 4 本身可以作为一棵子树，且二叉树中有多个节点 4：



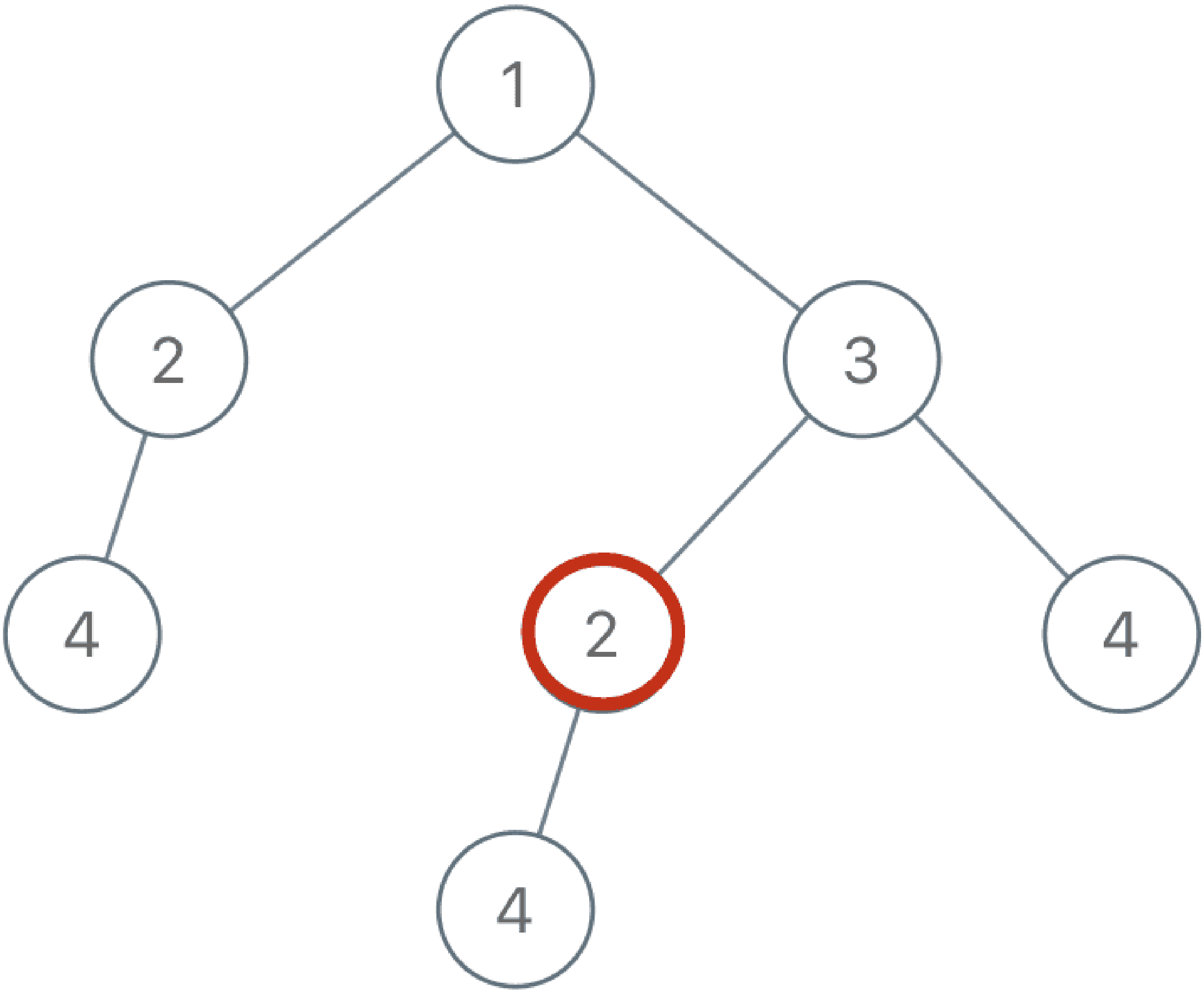
类似的，还存在两棵以 2 为根的重复子树：



那么，我们返回的 `List` 中就应该有两个 `TreeNode`，值分别为 4 和 2（具体是哪个节点都无所谓）。

这题咋做呢？**还是老套路，先思考，对于某一个节点，它应该做什么。**

比如说，你站在图中这个节点 2 上：



如果你想知道以自己为根的子树是不是重复的，是否应该被加入结果列表中，你需要知道什么信息？

你需要知道以下两点：

1、以我为根的这棵二叉树（子树）长啥样？

2、以其他节点为根的子树都长啥样？

这就叫知己知彼嘛，我得知道自己长啥样，还得知道别人长啥样，然后才能知道有没有人跟我重复，对不对？

————————————————

应合作方要求，本文不便在此发布，请扫码关注回复关键词「二叉树」查看：

二叉树的序列化，就那几个框架，枯燥至极

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong



学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

297. 二叉树的序列化和反序列化（困难）

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

JSON 的运用非常广泛，比如我们经常将变成语言中的结构体序列化成 JSON 字符串，存入缓存或者通过网络发送给远端服务，消费者接受 JSON 字符串然后进行反序列化，就可以得到原始数据了。这就是「序列化」和「反序列化」的目的，以某种固定格式组织字符串，使得数据可以独立于编程语言。

那么假设现在有一棵用 Java 实现的二叉树，我想把它序列化字符串，然后用 C++ 读取这棵并还原这棵二叉树的结构，怎么办？这就需要对二叉树进行「序列化」和「反序列化」了。

本文会用前序、中序、后序遍历的方式来序列化和反序列化二叉树，进一步，还会用迭代式的层级遍历来解决这个问题。

接下来就用二叉树的遍历框架来给你看看二叉树到底能玩出什么骚操作。

一、题目描述

力扣第 297 题「二叉树的序列化与反序列化」就是给你输入一棵二叉树的根节点 `root`，要求你实现如下一个类：

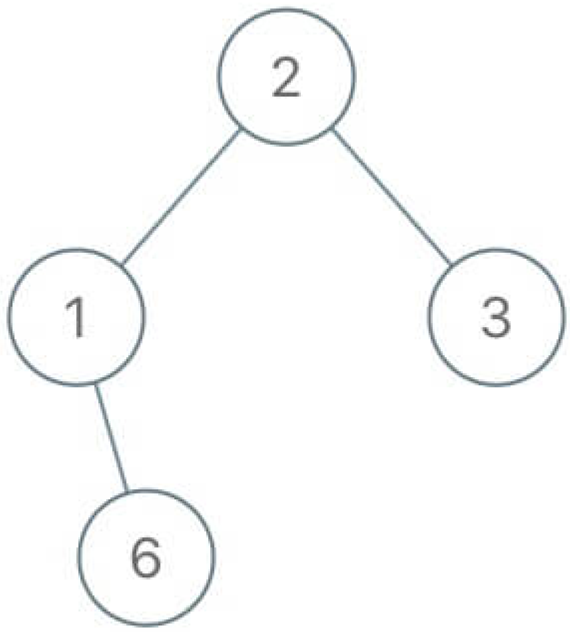
```java
public class Codec {

    // 把一棵二叉树序列化成字符串
    public String serialize(TreeNode root) {}

    // 把字符串反序列化成二叉树
    public TreeNode deserialize(String data) {}
}
```

我们可以用 `serialize` 方法将二叉树序列化成字符串，用 `deserialize` 方法将序列化的字符串反序列化成二叉树，至于以什么格式序列化和反序列化，这个完全由你决定。

比如说输入如下这样一棵二叉树：



serialize 方法也许会把它序列化成字符串 2,1,#,6,3,#,#，其中 # 表示 null 指针，那么把这个字符串再输入 deserialize 方法，依然可以还原出这棵二叉树。也就是说，这两个方法会成对儿使用，你只要保证他俩能够自洽就行了。

想象一下，二叉树结该是一个二维平面内的结构，而序列化出来的字符串是一个线性的一维结构。**所谓的序列化不过就是把结构化的数据「打平」，其实就是在考察二叉树的遍历方式。**

二叉树的遍历方式有哪些？递归遍历方式有前序遍历，中序遍历，后序遍历；迭代方式一般是层级遍历。本文就把这些方式都尝试一遍，来实现 serialize 方法和 deserialize 方法。

二、前序遍历解法

前文 学习数据结构和算法的框架思维 说过了二叉树的几种遍历方式，前序遍历框架如下：

```
void traverse(TreeNode root) {
    if (root == null) return;

    // 前序遍历的代码

    traverse(root.left);
    traverse(root.right);
}
```

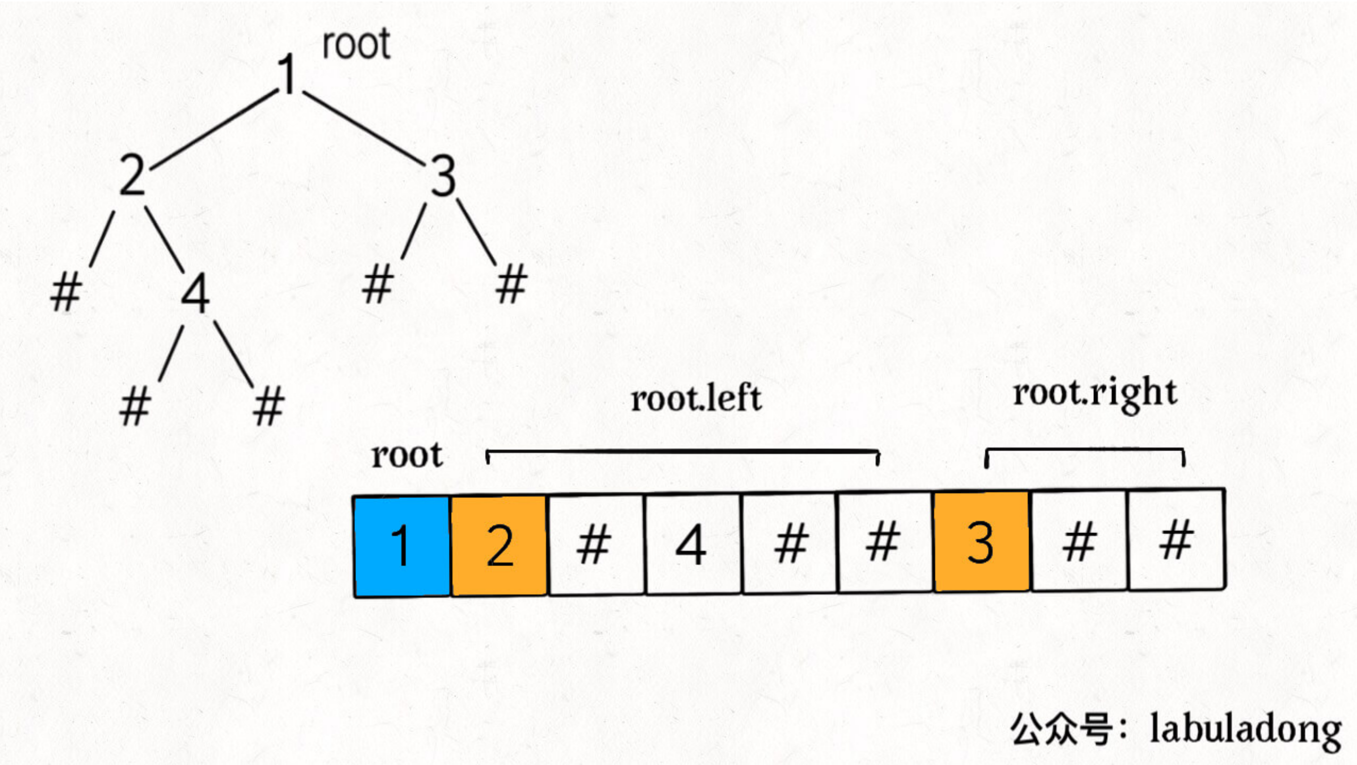
真的很简单，在递归遍历两棵子树之前写的代码就是前序遍历代码，那么请你看一看如下伪码：

```java
LinkedList<Integer> res;
void traverse(TreeNode root) {
    if (root == null) {
        // 暂且用数字 -1 代表空指针 null
        res.addLast(-1);
        return;
    }

    /****** 前序遍历位置 ******/
    res.addLast(root.val);
    /***********************/

    traverse(root.left);
    traverse(root.right);
}
```

调用 traverse 函数之后，你是否可以立即想出这个 res 列表中元素的顺序是怎样的？比如如下二叉树（# 代表空指针 null），可以直观看出前序遍历做的事情：



那么 res = [1,2,-1,4,-1,-1,3,-1,-1]，这就是将二叉树「打平」到了一个列表中，其中 -1 代表 null。

那么，将二叉树打平到一个字符串中也是完全一样的：

```java
// 代表分隔符的字符
String SEP = ",";
// 代表 null 空指针的字符
String NULL = "#";
// 用于拼接字符串
```

```java
StringBuilder sb = new StringBuilder();

/* 将二叉树打平为字符串 */
void traverse(TreeNode root, StringBuilder sb) {
    if (root == null) {
        sb.append(NULL).append(SEP);
        return;
    }

    /****** 前序遍历位置 ******/
    sb.append(root.val).append(SEP);
    /***********************/

    traverse(root.left, sb);
    traverse(root.right, sb);
}
```

`StringBuilder` 可以用于高效拼接字符串，所以也可以认为是一个列表，用 `,` 作为分隔符，用 `#` 表示空指针 null，调用完 `traverse` 函数后，`StringBuilder` 中的字符串应该是 `1,2,#,4,#,#,3,#,#,`。

至此，我们已经可以写出序列化函数 `serialize` 的代码了：

```java
String SEP = ",";
String NULL = "#";

/* 主函数，将二叉树序列化为字符串 */
String serialize(TreeNode root) {
    StringBuilder sb = new StringBuilder();
    serialize(root, sb);
    return sb.toString();
}

/* 辅助函数，将二叉树存入 StringBuilder */
void serialize(TreeNode root, StringBuilder sb) {
    if (root == null) {
        sb.append(NULL).append(SEP);
        return;
    }

    /****** 前序遍历位置 ******/
    sb.append(root.val).append(SEP);
    /***********************/

    serialize(root.left, sb);
    serialize(root.right, sb);
}
```

现在，思考一下如何写 `deserialize` 函数，将字符串反过来构造二叉树。

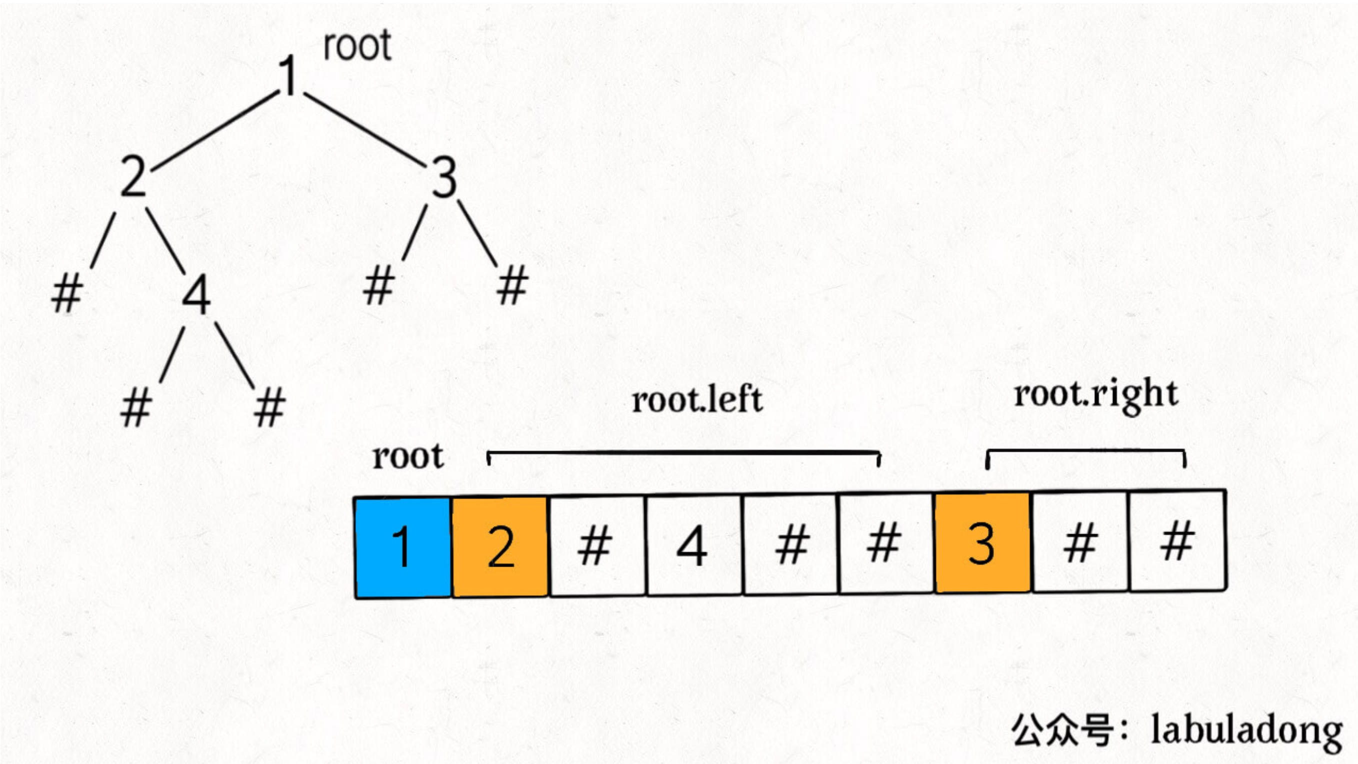
首先我们可以把字符串转化成列表：

```
String data = "1,2,#,4,#,#,3,#,#,";
String[] nodes = data.split(",");
```

这样，nodes 列表就是二叉树的前序遍历结果，问题转化为：如何通过二叉树的前序遍历结果还原一棵二叉树？

> PS：一般语境下，单单前序遍历结果是不能还原二叉树结构的，因为缺少空指针的信息，至少要得到前、中、后序遍历中的两种才能还原二叉树。但是这里的 node 列表包含空指针的信息，所以只使用 node 列表就可以还原二叉树。

根据我们刚才的分析，nodes 列表就是一棵打平的二叉树：



那么，反序列化过程也是一样，**先确定根节点 root，然后遵循前序遍历的规则，递归生成左右子树即可**：

应合作方要求，本文不便在此发布，请扫码关注回复关键词「序列化」查看：

美团面试官：你对后序遍历一无所知

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

1373. 二叉搜索子树的最大键值和（困难）

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

其实二叉树的题目真的不难，无非就是前中后序遍历框架来回倒嘛，但是对于有的题目，不同的遍历顺序时间复杂度不同。

之前面试美团，就遇到一道二叉树算法题，当时我是把解法写出来了，面试官说如果用后序遍历，时间复杂度可以更低。

本文就来分析一道类似的题目，通过二叉树的后序遍历，来大幅降低算法的复杂度。

手把手刷二叉树第一期 说过**二叉树相关题目最核心的思路是明确当前节点需要做的事情是什么。**

我们再看看后序遍历的代码框架：

```
void traverse(TreeNode root) {
    traverse(root.left);
    traverse(root.right);
    /* 后序遍历代码的位置 */
    /* 在这里处理当前节点 */
}
```

看这个代码框架，你说后序遍历什么时候出现呢？

如果当前节点要做的事情需要通过左右子树的计算结果推导出来，就要用到后序遍历。

很多时候，后序遍历用得好，可以大幅提升算法效率。

我们今天就要讲一个经典的算法问题，可以直观地体会到这一点。

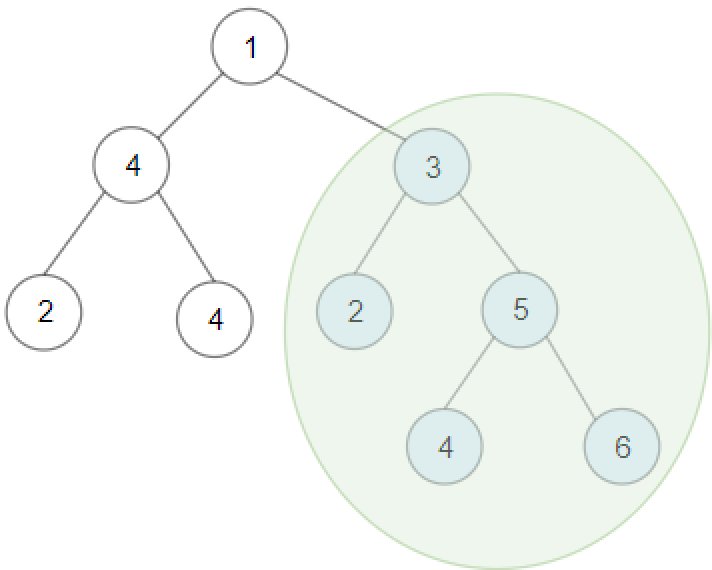
这是力扣第 1373 题「二叉搜索子树的最大键值和」，函数签名如下：

```
int maxSumBST(TreeNode root);
```

题目会给你输入一棵二叉树，这棵二叉树的子树中可能包含**二叉搜索树**对吧，请你找到**节点之和最大的**那棵二叉搜索树，返回它的节点值之和。

二叉搜索树（简写作 BST）的性质不用我多介绍了吧，简单说就是「左小右大」，对于每个节点，整棵左子树都比该节点的值小，整棵右子树都比该节点的值大。

比如题目给了这个例子：

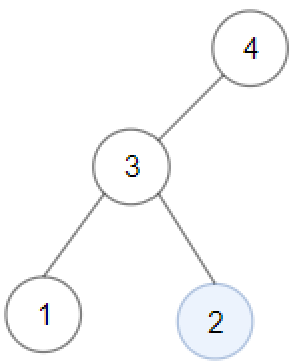


如果输入这棵二叉树，算法应该返回 20，也就是图中绿圈的那棵子树的节点值之和，因为它是一棵 BST，且节点之和最大。

那有的读者可能会问，根据 BST 的定义，有没有可能一棵二叉树中不存在 BST？

不会的，因为按照 BST 的定义，任何一个单独的节点肯定是 BST，也就是说，再不济，二叉树最下面的叶子节点肯定是 BST。

比如说如果输入下面这棵二叉树：



两个叶子节点 `1` 和 `2` 就是 BST，比较一下节点之和，算法应该返回 2。

好了，到这里，题目应该解释地很清楚了，下面我们来分析一下这道题应该怎么做。

刚才说了，**二叉树相关题目最核心的思路是明确当前节点需要做的事情是什么。**

那么我们想计算子树中 BST 的最大和，站在当前节点的视角，需要做什么呢？

1、我肯定得知道左右子树是不是合法的 BST，如果这俩儿子有一个不是 BST，以我为根的这棵树肯定不会是 BST，对吧。

2、如果左右子树都是合法的 BST，我得瞅瞅左右子树加上自己还是不是合法的 BST 了。因为按照 BST 的定义，当前节点的值应该大于左子树的最大值，小于右子树的最小值，否则就破坏了 BST 的性质。

3、因为题目要计算最大的节点之和，如果左右子树加上我自己还是一棵合法的 BST，也就是说以我为根的整棵树是一棵 BST，那我需要知道我们这棵 BST 的所有节点值之和是多少，方便和别的 BST 争个高下，对吧。

根据以上三点，站在当前节点的视角，需要知道以下具体信息：

1、左右子树是否是 BST。

2、左子树的最大值和右子树的最小值。

3、左右子树的节点值之和。

只有知道了这几个值，我们才能满足题目的要求，后面我们会想方设法计算这些值。

现在可以尝试用伪码写出算法的大致逻辑：

```
// 全局变量，记录 BST 最大节点之和
int maxSum = 0;

/* 主函数 */
public int maxSumBST(TreeNode root) {
    traverse(root);
    return maxSum;
}

/* 遍历二叉树 */
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }

    /******* 前序遍历位置 *******/
    // 判断左右子树是不是 BST
    if (!isBST(root.left) || !isBST(root.right)) {
        goto next;
    }
    // 计算左子树的最大值和右子树的最小值
    int leftMax = findMax(root.left);
    int rightMin = findMin(root.right);
    // 判断以 root 节点为根的树是不是 BST
    if (root.val <= leftMax || root.val >= rightMin) {
        goto next;
    }
```

```java
        // 如果条件都符合，计算当前 BST 的节点之和
        int leftSum = findSum(root.left);
        int rightSum = findSum(root.right);
        int rootSum = leftSum + rightSum + root.val;
        // 计算 BST 节点的最大和
        this.maxSum = Math.max(maxSum, rootSum);
        /**************************/

        // 递归左右子树
        next:
        traverse(root.left);
        traverse(root.right);
    }

    /* 计算以 root 为根的二叉树的最大值 */
    int findMax(TreeNode root) {}

    /* 计算以 root 为根的二叉树的最小值 */
    int findMin(TreeNode root) {}

    /* 计算以 root 为根的二叉树的节点和 */
    int findSum(TreeNode root) {}

    /* 判断以 root 为根的二叉树是否是 BST */
    boolean isBST(TreeNode root) {}
```

这个代码逻辑应该是不难理解的，代码在前序遍历的位置把之前的分析都实现了一遍。

其中有四个辅助函数比较简单，我就不具体实现了，其中只有判断合法 BST 的函数稍有技术含量，前文 二叉搜索树操作集锦 写过，这里就不展开了。

稍作分析就会发现，这几个辅助函数都是递归函数，都要遍历输入的二叉树，外加 `traverse` 函数本身的递归，可以说是递归上加递归，所以这个解法的复杂度是非常高的。

但是根据刚才的分析，像 `leftMax`、`rootSum` 这些变量又都得算出来，否则无法完成题目的要求。

我们希望既算出这些变量，又避免辅助函数带来的额外复杂度，鱼和熊掌全都要！

其实是可以的，**只要把前序遍历变成后序遍历，让 `traverse` 函数把辅助函数做的事情顺便做掉。**

其他代码不变，我们让 `traverse` 函数做一些计算任务，返回一个数组：

```java
// 全局变量，记录 BST 最大节点之和
int maxSum = 0;

/* 主函数 */
public int maxSumBST(TreeNode root) {
    traverse(root);
    return maxSum;
}

// 函数返回 int[]{ isBST, min, max, sum}
```

```
int[] traverse(TreeNode root) {

    int[] left = traverse(root.left);
    int[] right = traverse(root.right);

    /******* 后序遍历位置 *******/
    // 通过 left 和 right 推导返回值
    // 并且正确更新 maxSum 变量
    /**************************/
}
```

`traverse(root)` 返回一个大小为 4 的 int 数组，我们暂且称它为 `res`，其中：

`res[0]` 记录以 `root` 为根的二叉树是否是 BST，若为 1 则说明是 BST，若为 0 则说明不是 BST；

`res[1]` 记录以 `root` 为根的二叉树所有节点中的最小值；

`res[2]` 记录以 `root` 为根的二叉树所有节点中的最大值；

`res[3]` 记录以 `root` 为根的二叉树所有节点值之和。

其实这就是把之前分析中说到的几个值放到了 `res` 数组中，**最重要的是，我们要试图通过 `left` 和 `right` 正确推导出 `res` 数组。**

直接看代码实现吧：

```
int[] traverse(TreeNode root) {
    // base case
    if (root == null) {
        return new int[] {
            1, Integer.MAX_VALUE, Integer.MIN_VALUE, 0
        };
    }

    // 递归计算左右子树
    int[] left = traverse(root.left);
    int[] right = traverse(root.right);

    /******* 后序遍历位置 *******/
    int[] res = new int[4];
    // 这个 if 在判断以 root 为根的二叉树是不是 BST
    if (left[0] == 1 && right[0] == 1 &&
        root.val > left[2] && root.val < right[1]) {
        // 以 root 为根的二叉树是 BST
        res[0] = 1;
        // 计算以 root 为根的这棵 BST 的最小值
        res[1] = Math.min(left[1], root.val);
        // 计算以 root 为根的这棵 BST 的最大值
        res[2] = Math.max(right[2], root.val);
        // 计算以 root 为根的这棵 BST 所有节点之和
        res[3] = left[3] + right[3] + root.val;
        // 更新全局变量
```

```
            maxSum = Math.max(maxSum, res[3]);
        } else {
            // 以 root 为根的二叉树不是 BST
            res[0] = 0;
            // 其他的值都没必要计算了，因为用不到
        }
        /**************************/

        return res;
    }
```

这样，这道题就解决了，traverse 函数在遍历二叉树的同时顺便把之前辅助函数做的事情都做了，避免了在递归函数中调用递归函数，时间复杂度只有 O(N)。

你看，这就是后序遍历的妙用，相对前序遍历的解法，现在的解法不仅效率高，而且代码量少，比较优美。

那肯定有读者问，后序遍历这么好，是不是就应该尽可能多用后序遍历？

其实也不是，主要是看题目，就好比 BST 的中序遍历是有序的一样。

这道题为什么用后序遍历呢，因为我们需要的这些变量都是可以通过后序遍历得到的。

你计算以 root 为根的二叉树的节点之和，是不是可以通过左右子树的和加上 root.val 计算出来？

你计算以 root 为根的二叉树的最大值/最小值，是不是可以通过左右子树的最大值/最小值和 root.val 比较出来？

你判断以 root 为根的二叉树是不是 BST，是不是得先判断左右子树是不是 BST？是不是还得看看左右子树的最大值和最小值？

文章开头说过，**如果当前节点要做的事情需要通过左右子树的计算结果推导出来，就要用到后序遍历。**

因为以上几点都可以通过后序遍历的方式计算出来，所以这道题使用后序遍历肯定是最高效的。

以我的刷题经验，我们要尽可能避免递归函数中调用其他递归函数，如果出现这种情况，大概率是代码实现有瑕疵，可以进行类似本文的优化来避免递归套递归。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

东哥带你刷二叉树（总结篇）

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong



学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

104. 二叉树的最大深度（简单）

543. 二叉树的直径（简单）

144. 二叉树的前序遍历（简单）

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

公众号历史文章都是按照 学习数据结构和算法的框架思维 提出的框架思维来构建的，其中着重强调了二叉树题目的重要性，所以我写了 6 篇手把手刷二叉树系列文章：

- 东哥手把手带你刷二叉树（第一期）
- 东哥手把手带你刷二叉树（第二期）
- 东哥手把手带你刷二叉树（第三期）
- 东哥手把手带你刷二叉搜索树（第一期）
- 东哥手把手带你刷二叉搜索树（第二期）
- 东哥手把手带你刷二叉搜索树（第三期）

本文是一个总纲，对上述文章进行总结，无论你是否看过上述几篇文章，看完本文再去看它们都会对二叉树和递归有更深的认识。

另外，本文中会用题目来举例，但都是最最简单的题目，所以不用担心自己看不懂。我可以帮你从最简单的问题中提炼出所有二叉树题目的共性，并将这些思维反手用到 动态规划，回溯算法，分治算法，图论算法 中去，这也是我一直强调框架思维的原因。

深入理解前中后序

之前二叉树的文章，总有读者说看不出解法应该用前序中序还是后序，其实原因是你对前中后序的理解还不到位，这里我简单解释一下。

首先，先回顾一下 学习数据结构和算法的框架思维 中说到的二叉树遍历框架：

```java
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序位置
    traverse(root.left);
    // 中序位置
    traverse(root.right);
    // 后序位置
}
```

先不管所谓前中后序，单看这段代码是什么？

其实就是一个能够遍历二叉树所有节点的一个函数，和你遍历数组或者链表本质上没有区别：

```java
/* 迭代遍历数组 */
void traverse(int[] arr) {
    for (int i = 0; i < arr.length; i++) {

    }
}

/* 递归遍历数组 */
void traverse(int[] arr, int i) {
    if (i == arr.length) {
        return;
    }
    // 前序位置
    traverse(arr, i + 1);
    // 后序位置
}

/* 迭代遍历单链表 */
void traverse(ListNode head) {
    for (ListNode p = head; p != null; p = p.next) {

    }
}

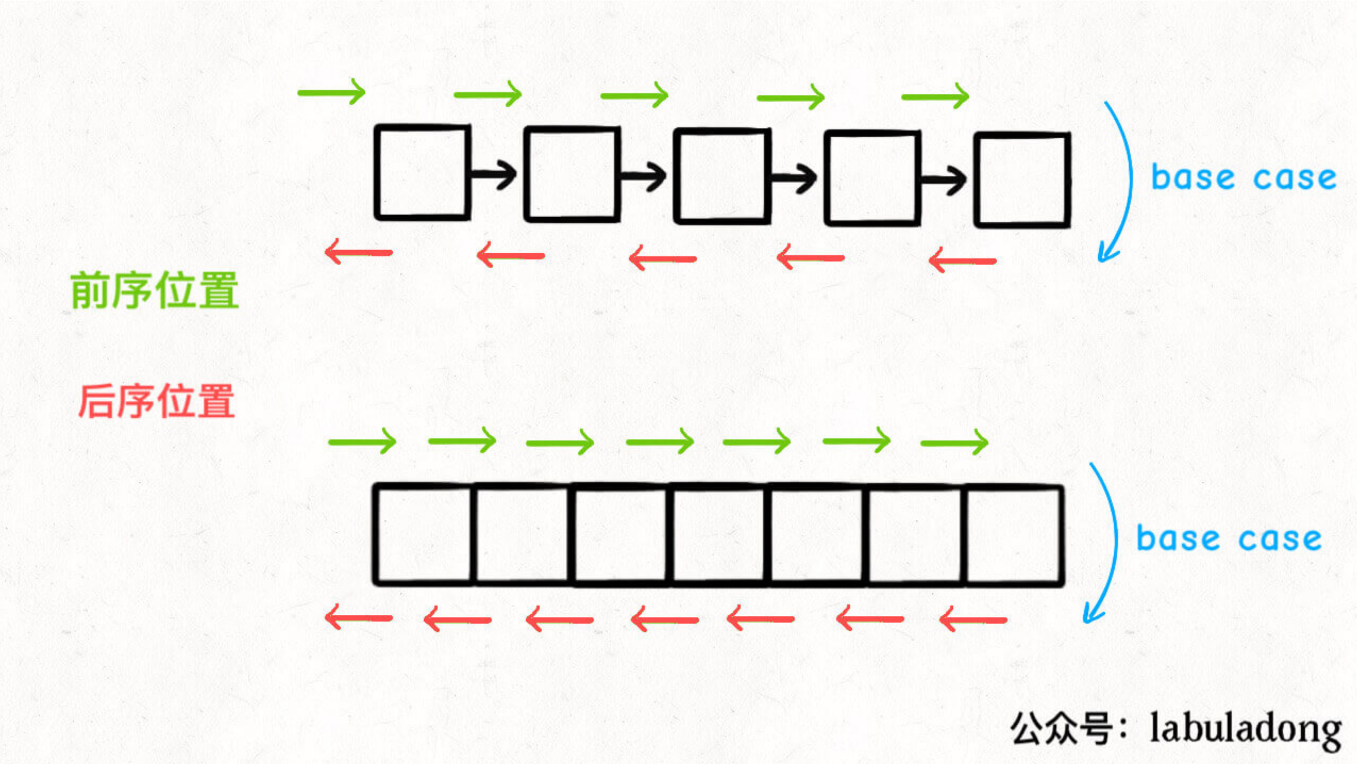
/* 递归遍历单链表 */
void traverse(ListNode head) {
    if (head == null) {
        return;
    }
    // 前序位置
    traverse(head.next);
    // 后序位置
}
```

单链表和数组的遍历可以是迭代的，也可以是递归的，二叉树这种结构无非就是二叉链表，不过没办法简单改写成迭代形式，所以一般说二叉树的遍历框架都是指递归的形式。

你也注意到了，只要是递归形式的遍历，都会有一个前序和后序位置，分别在递归之前和之后。

所谓前序位置，就是刚进入一个节点（元素）的时候，后序位置就是即将离开一个节点（元素）的时候。

你把代码写在不同位置，代码执行的时机也不同：



比如说，如果让你**倒序打印**一条单链表上所有节点的值，你怎么搞？

实现方式当然有很多，但如果你对递归的理解足够透彻，可以利用后序位置：

```
/* 递归遍历单链表，倒序打印链表元素 */
void traverse(ListNode head) {
    if (head == null) {
        return;
    }
    traverse(head.next);
    // 后序位置
    print(head.val);
}
```

结合上面那张图，你应该知道为什么这段代码能够倒序打印单链表了吧，本质上是利用递归的堆栈帮你实现了倒序遍历的效果。

那么说回二叉树也是一样的，只不过多了一个中序位置罢了。

这里我强调一个初学者经常犯的错误：因为教科书里只会问你前中后序遍历结果分别是什么，所以对于一个只上过大学数据结构课程的人来说，他大概以为二叉树的前中后序只不过对应三种顺序不同的

`List<Integer>` 列表。

但是我想说，**前中后序是遍历二叉树过程中处理每一个节点的三个特殊时间点**，绝不仅仅是三个顺序不同的 List：

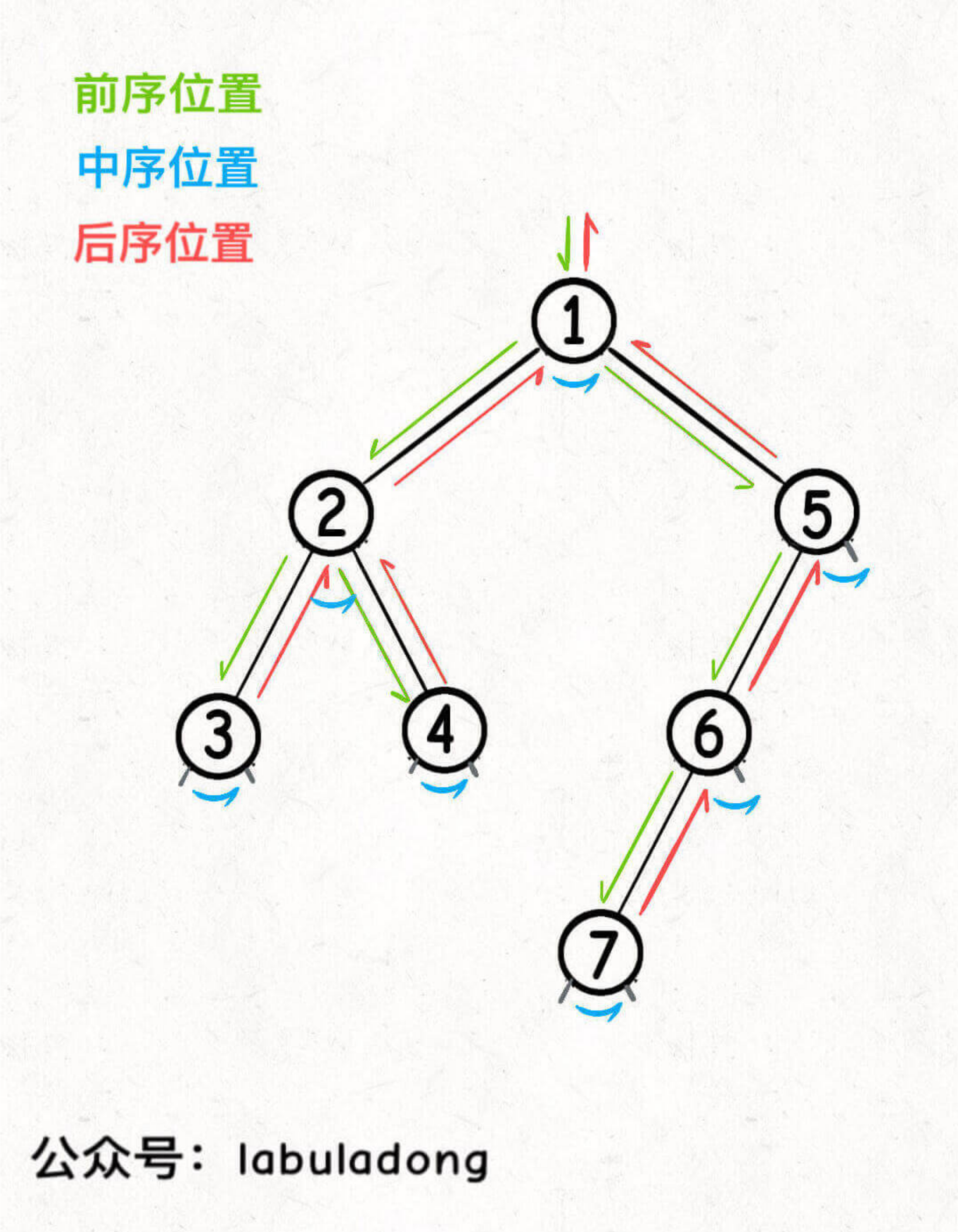
前序位置的代码在刚刚进入一个二叉树节点的时候执行；

后序位置的代码在将要离开一个二叉树节点的时候执行；

中序位置的代码在一个二叉树节点左子树都遍历完，即将开始遍历右子树的时候执行。

你注意本文的用词，我一直说前中后序「位置」，就是要和大家常说的前中后序「遍历」有所区别：你可以在前序位置写代码往一个 List 里面塞元素，那最后可以得到前序遍历结果；但并不是说你就不可以写更复杂的代码做更复杂的事。

画成图，前中后序三个位置在二叉树上是这样：



你可以发现**每个节点都有「唯一」属于自己的前中后序位置**，所以我说前中后序遍历是遍历二叉树过程中处理每一个节点的三个特殊时间点。

这里你也可以理解为什么多叉树没有中序位置，因为二叉树的每个节点只会进行唯一一次左子树切换右子树，而多叉树节点可能有很多子节点，会多次切换子树去遍历，所以多叉树节点没有「唯一」的中序遍历位置。

说了这么多基础的，就是要帮你对二叉树建立正确的认识，然后你会发现：**二叉树的所有问题，就是让你在前中后序位置注入巧妙的代码逻辑，去达到自己的目的。**

这也就是之前 6 篇手把手刷二叉树文章的核心思想：你只需要思考每一个节点应该做什么，其他的不用你管，抛给二叉树遍历框架，递归会对所有节点做相同的操作。

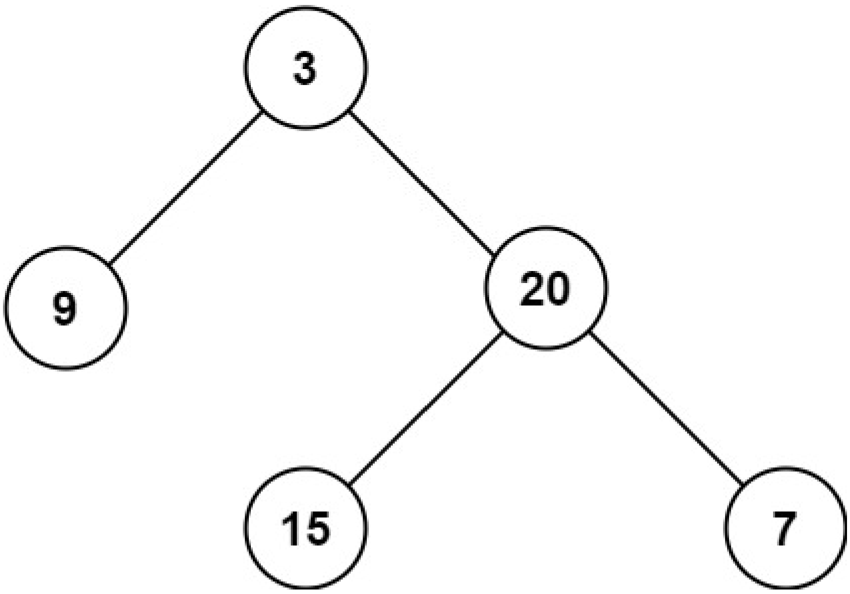
你也可以看到，图论算法基础 把二叉树的遍历框架扩展到了图，并以遍历为基础实现了图论的各种经典算法，不过这是后话，本文就不多说了。

两种解题思路

前文 我的算法学习心得 说过，**二叉树题目的递归解法可以分两类思路，第一类是遍历一遍二叉树得出答案，第二类是通过分解问题计算出答案，这两类思路分别对应着 回溯算法核心框架 和 动态规划核心框架**。

当时我是用二叉树的最大深度这个问题来举例，重点在于把这两种思路和动态规划和回溯算法进行对比，而本文的重点在于分析这两种思路如何解决二叉树的题目。

力扣第 104 题「二叉树的最大深度」就是最大深度的题目，所谓最大深度就是根节点到「最远」叶子节点的最长路径上的节点数，比如输入这棵二叉树，算法应该返回 3：



你做这题的思路是什么？显然遍历一遍二叉树，用一个外部变量记录每个节点所在的深度，取最大值就可以得到最大深度，**这就是遍历二叉树计算答案的思路**。

解法代码如下：

```
// 记录最大深度
int res = 0;
// 记录遍历到的节点的深度
int depth = 0;

// 主函数
int maxDepth(TreeNode root) {
    traverse(root);
    return res;
}
```

```java
// 二叉树遍历框架
void traverse(TreeNode root) {
    if (root == null) {
        // 到达叶子节点，更新最大深度
        res = Math.max(res, depth);
        return;
    }
    // 前序位置
    depth++;
    traverse(root.left);
    traverse(root.right);
    // 后序位置
    depth--;
}
```

这个解法应该很好理解，但为什么需要在前序位置增加 `depth`，在后序位置减小 `depth`?

因为前面说了，前序位置是进入一个节点的时候，后序位置是离开一个节点的时候，`depth` 记录当前递归到的节点深度，所以要这样维护。

当然，你也很容易发现一棵二叉树的最大深度可以通过子树的最大高度推导出来，**这就是分解问题计算答案的思路。**

解法代码如下：

```java
// 定义：输入根节点，返回这棵二叉树的最大深度
int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    // 利用定义，计算左右子树的最大深度
    int leftMax = maxDepth(root.left);
    int rightMax = maxDepth(root.right);
    // 整棵树的最大深度等于左右子树的最大深度取最大值，
    // 然后再加上根节点自己
    int res = Math.max(leftMax, rightMax) + 1;

    return res;
}
```

只要明确递归函数的定义，这个解法也不难理解，但为什么主要的代码逻辑集中在后序位置?

因为这个思路正确的核心在于，你确实可以通过子树的最大高度推导出原树的高度，所以当然要首先利用递归函数的定义算出左右子树的最大深度，然后推出原树的最大深度，主要逻辑自然放在后序位置。

如果你理解了最大深度这个问题的两种思路，**那么我们再回头看看最基本的二叉树前中后序遍历**，就比如算前序遍历结果吧。

我们熟悉的解法就是用「遍历」的思路，我想应该没什么好说的：

```java
List<Integer> res = new LinkedList<>();

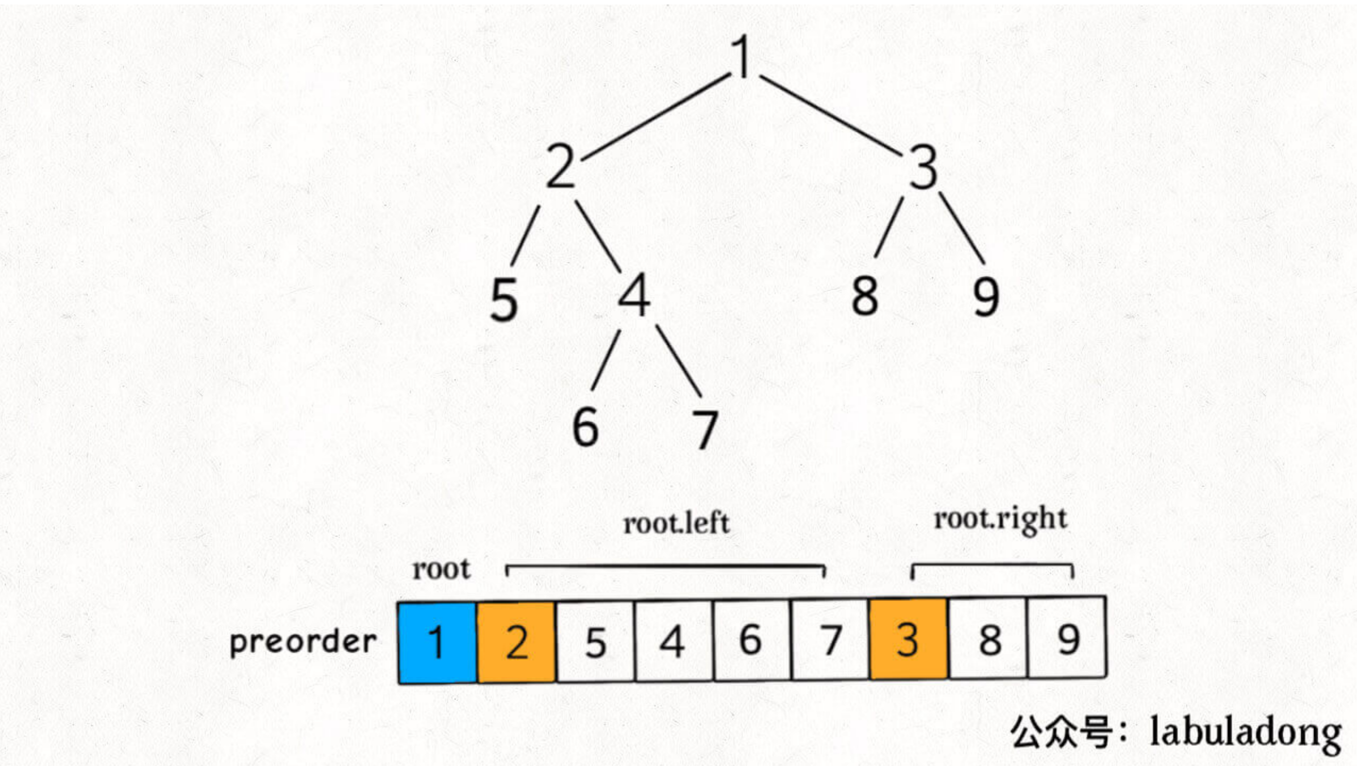
// 返回前序遍历结果
List<Integer> preorderTraverse(TreeNode root) {
    traverse(root);
    return res;
}

// 二叉树遍历函数
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序位置
    res.add(root.val);
    traverse(root.left);
    traverse(root.right);
}
```

但你是否能够用「分解问题」的思路，来计算前序遍历的结果？

换句话说，不要用像 `traverse` 这样的辅助函数和任何外部变量，单纯用题目给的 `preorderTraverse` 函数递归解题，你会不会？

我们知道前序遍历的特点是，根节点的值排在首位，接着是左子树的前序遍历结果，最后是右子树的前序遍历结果：



那这不就可以分解问题了么，**一棵二叉树的前序遍历分解成了根节点和左右子树的前序遍历结果。**

所以，你可以这样实现前序遍历算法：

```java
// 定义：输入一棵二叉树的根节点，返回这棵树的前序遍历结果
List<Integer> preorderTraverse(TreeNode root) {
    List<Integer> res = new LinkedList<>();
    if (root == null) {
        return res;
    }
    // 前序遍历的结果，root.val 在第一个
    res.add(root.val);
    // 利用函数定义，后面接着左子树的前序遍历结果
    res.addAll(preorderTraverse(root.left));
    // 利用函数定义，最后接着右子树的前序遍历结果
    res.addAll(preorderTraverse(root.right));
}
```

中序和后序遍历也是类似的，只要把 `add(root.val)` 放到中序和后序对应的位置就行了。

这个解法短小精干，但为什么不常见呢？

一个原因是**这个算法的复杂度不好把控**，比较依赖语言特性。

Java 的话无论 ArrayList 还是 LinkedList，`addAll` 方法的复杂度都是 O(N)，所以总体的最坏时间复杂度会达到 O(N^2)，除非你自己实现一个复杂度为 O(1) 的 `addAll` 方法，底层用链表的话并不是不可能。

当然，最主要的原因还是因为教科书上从来没有这么教过……

上文举了两个简单的例子，但还有不少二叉树的题目是可以同时使用两种思路来思考和求解的，这就要靠你自己多去练习和思考，不要仅仅满足于一种熟悉的解法思路。

综上，遇到一道二叉树的题目时的通用思考过程是：

是否可以通过遍历一遍二叉树得到答案？如果不能的话，是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案？

我的刷题插件 更新了所有值得一做的二叉树题目思路，全部归类为上述两种思路，你如果按照插件提供的思路解法过一遍二叉树的所有题目，不仅可以完全掌握递归思维，而且可以更容易理解高级的算法：

669. 修剪二叉搜索树 思路

难度 中等 451

[x]

基本思路

前文 手把手刷二叉树 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，显然这道题需要用到「分解问题」的思维。

明确了递归函数的定义之后进行思考，如果一个节点的值没有落在 `[lo, hi]` 中，有两种情况：

1、 `root.val < lo` ，这种情况下 `root` 节点本身和 `root` 的左子树全都是小于 `lo` 的，都需要被剪掉。

2、 `root.val > hi` ，这种情况下 `root` 节点本身和 `root` 的右子树全都是大于 `hi` 的，都需要被剪掉。

标签：二叉搜索树

解法代码

Copy

```java
class Solution {
    // 定义：删除 BST 中小于 low 和大于 high 的所有节点，返回结果 BST
    public TreeNode trimBST(TreeNode root, int low, int high) {
        if (root == null) return null;

        if (root.val < low) {
```

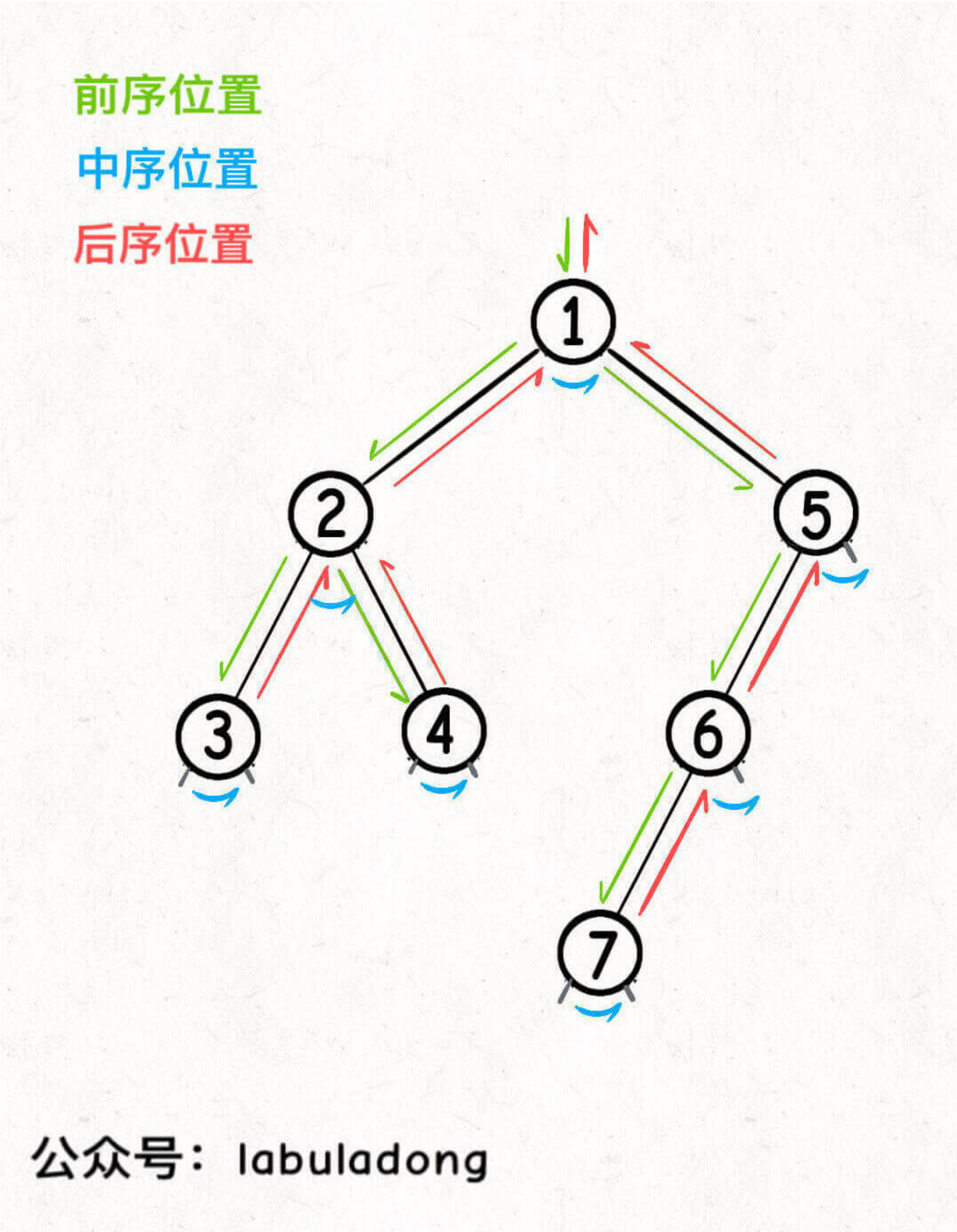
后序位置的特殊之处

说后序位置之前，先简单说下中序和前序。

中序位置主要用在 BST 场景中，你完全可以把 BST 的中序遍历认为是遍历有序数组。

前序位置本身其实没有什么特别的性质，之所以你发现好像很多题都是在前序位置写代码，实际上是因为我们习惯把那些对前中后序位置不敏感的代码写在前序位置罢了。

接下来主要说下后序位置，和前序位置对比，发现前序位置的代码执行是自顶向下的，而后序位置的代码执行是自底向上的：



这不奇怪，因为本文开头就说了前序位置是刚刚进入节点的时刻，后序位置是即将离开节点的时刻。

但这里面大有玄妙，意味着前序位置的代码只能从函数参数中获取父节点传递来的数据，而后序位置的代码不仅可以获取参数数据，还可以获取到子树通过函数返回值传递回来的数据。

举具体的例子，现在给你一棵二叉树，我问你两个简单的问题：

1、如果把根节点看做第 1 层，如何打印出每一个节点所在的层数？

2、如何打印出每个节点的左右子树各有多少节点？

第一个问题可以这样写代码：

```
// 二叉树遍历函数
void traverse(TreeNode root, int level) {
    if (root == null) {
        return;
    }
    // 前序位置
    printf("节点 %s 在第 %d 层", root, level);
    traverse(root.left, level + 1);
    traverse(root.right, level + 1);
}

// 这样调用
traverse(root, 1);
```

第二个问题可以这样写代码：

```
// 定义：输入一棵二叉树，返回这棵二叉树的节点总数
int count(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftCount = count(root.left);
    int rightCount = count(root.right);
    // 后序位置
    printf("节点 %s 的左子树有 %d 个节点，右子树有 %d 个节点",
            root, leftCount, rightCount);

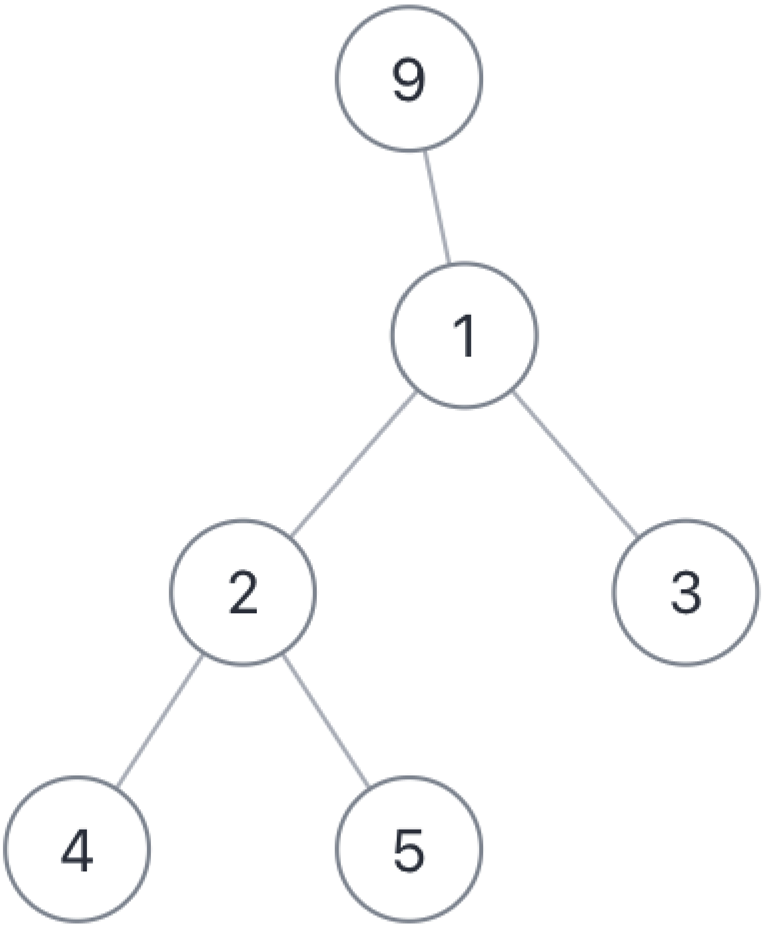
    return leftCount + rightCount + 1;
}
```

结合这两个简单的问题，你品味一下后序位置的特点，只有后序位置才能通过返回值获取子树的信息。

那么换句话说，一旦你发现题目和子树有关，那大概率要给函数设置合理的定义和返回值，在后序位置写代码了。

接下来看下后序位置是如何在实际的题目中发挥作用的，简单聊下力扣第 543 题「二叉树的直径」，让你计算一棵二叉树的最长直径长度。

所谓二叉树的「直径」长度，就是任意两个结点之间的路径长度。最长「直径」并不一定要穿过根结点，比如下面这棵二叉树：



它的最长直径是 3，即 [4,2,1,3] 或者 [5,2,1,3] 这两条「直径」的长度。

解决这题的关键在于，**每一条二叉树的「直径」长度，就是一个节点的左右子树的最大深度之和。**

现在让我求整棵树中的最长「直径」，那直截了当的思路就是遍历整棵树中的每个节点，然后通过每个节点的左右子树的最大深度算出每个节点的「直径」，最后把所有「直径」求个最大值即可。

最大深度的算法我们刚才实现过了，上述思路就可以写出以下代码：

```
// 记录最大直径的长度
int maxDiameter = 0;

public int diameterOfBinaryTree(TreeNode root) {
    // 对每个节点计算直径，求最大直径
    traverse(root);
    return maxDiameter;
}

// 遍历二叉树
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 对每个节点计算直径
```

```java
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        int myDiameter = leftMax + rightMax;
        // 更新全局最大直径
        maxDiameter = Math.max(maxDiameter, myDiameter);

        traverse(root.left);
        traverse(root.right);
    }

    // 计算二叉树的最大深度
    int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        return 1 + Math.max(leftMax, rightMax);
    }
```

这个解法是正确的，但是运行时间很长，原因也很明显，`traverse` 遍历每个节点的时候还会调用递归函数 `maxDepth`，而 `maxDepth` 是要遍历子树的所有节点的，所以最坏时间复杂度是 O(N^2)。

这就出现了刚才探讨的情况，**前序位置无法获取子树信息，所以只能让每个节点调用 `maxDepth` 函数去算子树的深度**。

那如何优化？我们应该把计算「直径」的逻辑放在后序位置，准确说应该是放在 `maxDepth` 的后序位置，因为 `maxDepth` 的后序位置是知道左右子树的最大深度的。

所以，稍微改一下代码逻辑即可得到更好的解法：

```java
    // 记录最大直径的长度
    int maxDiameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        maxDepth(root);
        return maxDiameter;
    }

    int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        // 后序位置顺便计算最大直径
        int myDiameter = leftMax + rightMax;
        maxDiameter = Math.max(maxDiameter, myDiameter);

        return 1 + Math.max(leftMax, rightMax);
    }
```

这下时间复杂度只有 `maxDepth` 函数的 O(N) 了。

讲到这里，照应一下前文：遇到子树问题，首先想到的是给函数设置返回值，然后在后序位置做文章。

反过来，如果你写出了类似一开始的那种递归套递归的解法，大概率也需要反思是不是可以通过后序遍历优化了。

我的刷题插件对于这类考察后序遍历的题目也有特殊的说明，并且会给出前置题目，帮助你由浅入深理解这类题目：

124. 二叉树中的最大路径和 思路

难度 困难 1340

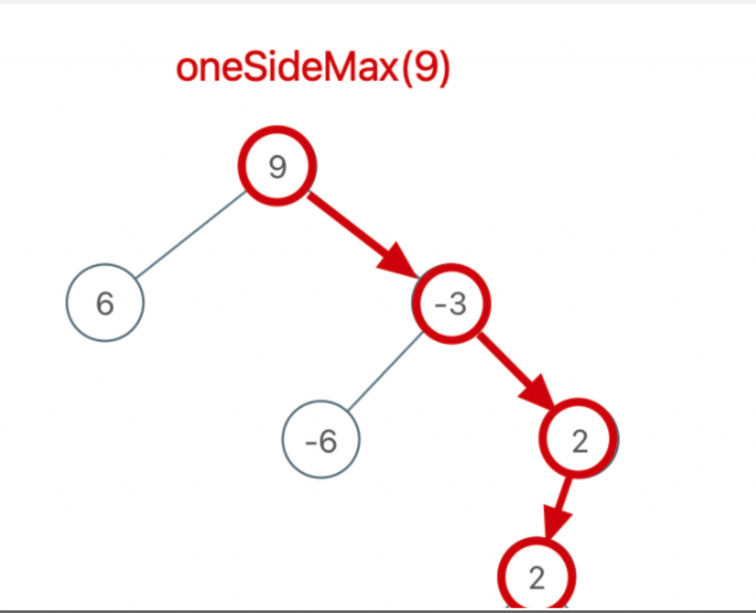
[x]

基本思路

前文 手把手带你刷二叉树 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，显然这道题需要用到「分解问题」的思维。

这题需要巧用二叉树的后序遍历，可以先去做一下 543. 二叉树的直径 和 366. 寻找二叉树的叶子节点。

`oneSideMax` 函数和上述几道题中都用到的 `maxDepth` 函数非常类似，只不过 `maxDepth` 计算最大深度，`oneSideMax` 计算「单边」最大路径和：



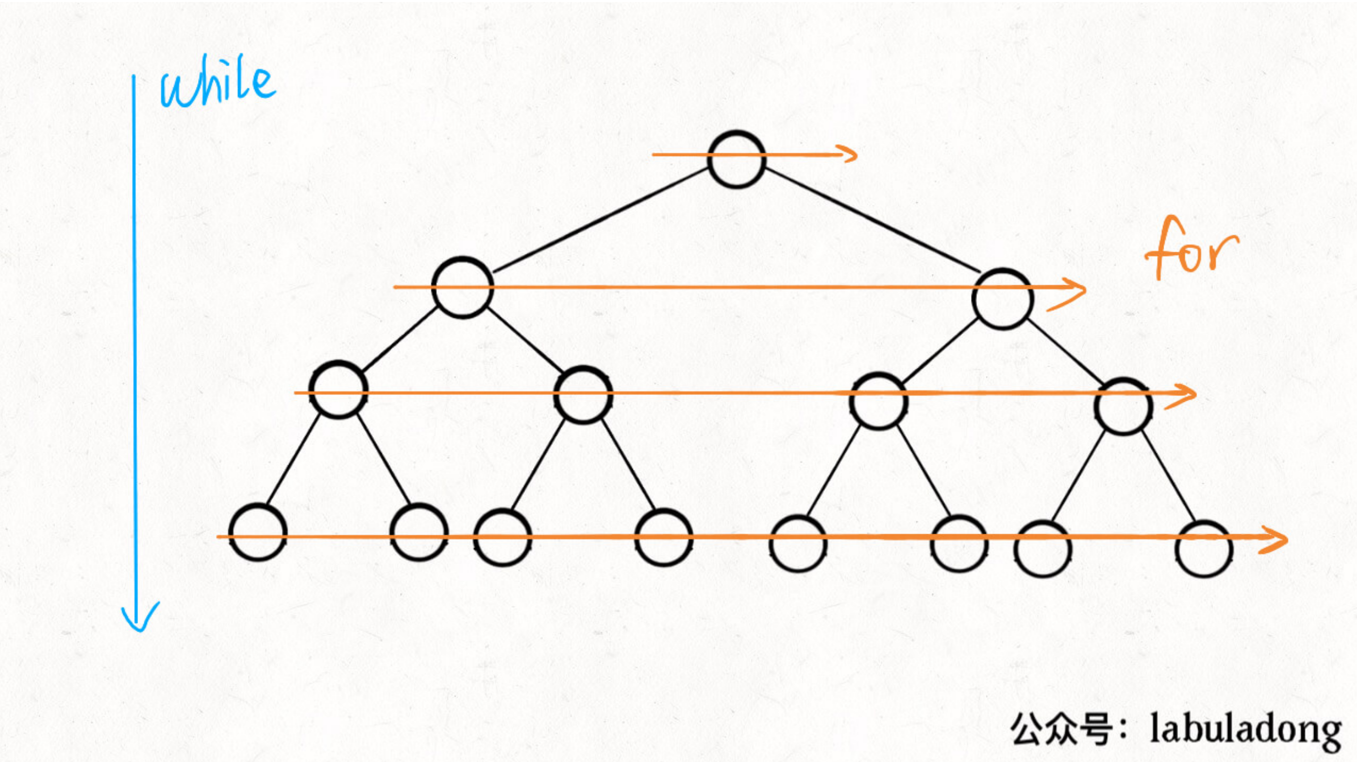
层序遍历

二叉树题型主要是用来培养递归思维的，而层序遍历属于迭代遍历，也比较简单，这里就过一下代码框架吧：

```java
// 输入一棵二叉树的根节点，层序遍历这棵二叉树
void levelTraverse(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    // 从上到下遍历二叉树的每一层
    while (!q.isEmpty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            // 将下一层节点放入队列
            if (cur.left != null) {
                q.offer(cur.left);
            }
            if (cur.right != null) {
                q.offer(cur.right);
            }
        }
    }
}
```

这里面 while 循环和 for 循环分管从上到下和从左到右的遍历：



前文 BFS 算法框架 就是从二叉树的层序遍历扩展出来的，常用于求无权图的**最短路径**问题。

当然这个框架还可以灵活修改，题目不需要记录层数（步数）时可以去掉上述框架中的 for 循环，比如前文 Dijkstra 算法 中计算加权图的最短路径问题，详细探讨了 BFS 算法的扩展。

值得一提的是，有些很明显需要用层序遍历技巧的二叉树的题目，也可以用递归遍历的方式去解决，而且技巧性会更强，非常考察你对前中后序的把控。

对于这类问题，[我的刷题插件](#)也会同时提供递归遍历和层序遍历的解法代码：

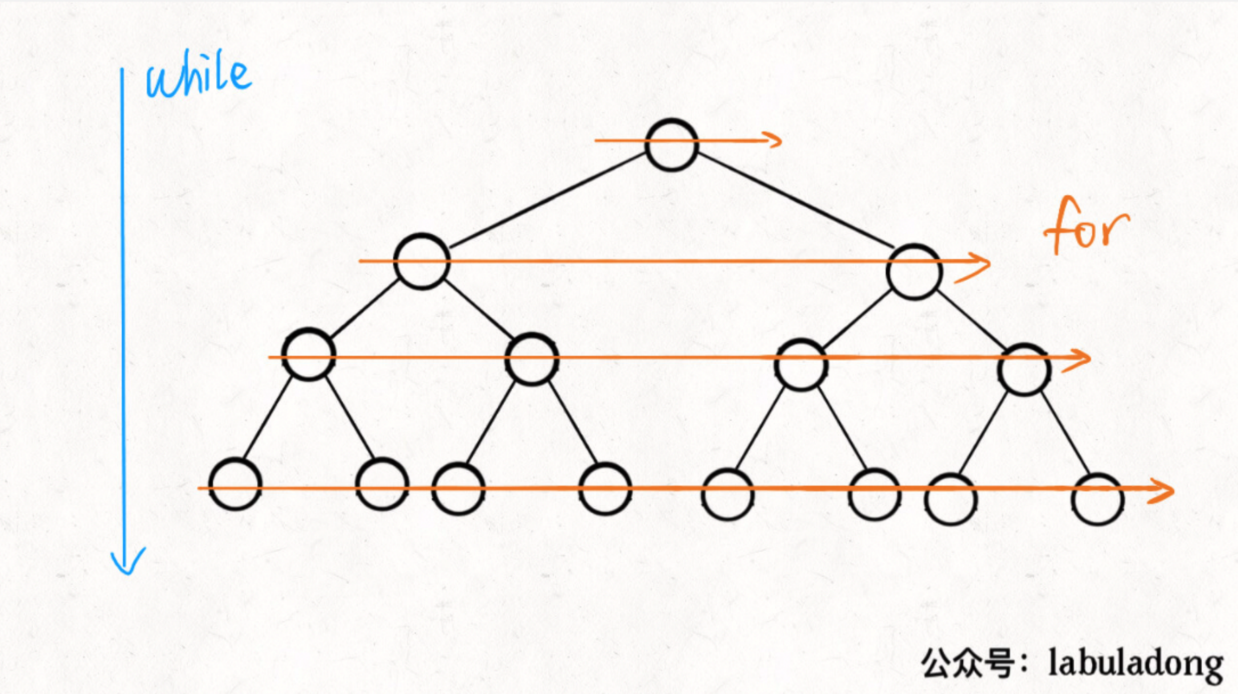
515. 在每个树行中找最大值 思路

难度 中等 162

[x]

基本思路

首先，这题肯定可以用 BFS 算法解决，for 循环里面判断最大值就行了：



当然，这题也可以用 DFS 来做，前文 [我的算法学习经验](#) 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，显然这道题需要用到「遍历」的思维。

遍历的过程中记录对应深度的最大节点值即可。

示例3：

好了，本文已经够长了，围绕前中后序位置算是把二叉树题目里的各种套路给讲透了，真正能运用出来多少，就需要你亲自刷题实践和思考了。

希望大家能探索尽可能多的解法，只要参透二叉树这种基本数据结构的原理，那么就很容易在学习其他高级算法的道路上找到抓手，打通回路，形成闭环（手动狗头）。

最后，我在不断完善刷题插件对二叉树系列题目的支持，在公众号后台回复关键词「**插件**」即可下载。

如果你有高质量的问题，可以 [点击这里](#) 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

2.2 二叉搜索树

BST 是一种特殊的二叉树，你只要记住它的两个主要特点：

1、左小右大，即每个节点的左子树都比当前节点的值小，右子树都比当前节点的值大。

2、中序遍历结果是有序的。

公众号标签：二叉搜索树

东哥带你刷二叉搜索树（第一期）

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

230. 二叉搜索树中第 K 小的元素（中等）

538. 二叉搜索树转化累加树（中等）

1038. BST 转累加树（中等）

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

前文手把手带你刷二叉树已经写了 第一期，第二期 和 第三期，今天写一篇二叉搜索树（Binary Search Tree，后文简写 BST）相关的文章，手把手带你刷 BST。

首先，BST 的特性大家应该都很熟悉了：

1、对于 BST 的每一个节点 `node`，左子树节点的值都比 `node` 的值要小，右子树节点的值都比 `node` 的值大。

2、对于 BST 的每一个节点 `node`，它的左侧子树和右侧子树都是 BST。

二叉搜索树并不算复杂，但我觉得它可以算是数据结构领域的半壁江山，直接基于 BST 的数据结构有 AVL 树，红黑树等等，拥有了自平衡性质，可以提供 logN 级别的增删查改效率；还有 B+ 树，线段树等结构都是基于 BST 的思想来设计的。

从做算法题的角度来看 BST，除了它的定义，还有一个重要的性质：BST 的中序遍历结果是有序的（升序）。

也就是说，如果输入一棵 BST，以下代码可以将 BST 中每个节点的值升序打印出来：

```
void traverse(TreeNode root) {
    if (root == null) return;
    traverse(root.left);
    // 中序遍历代码位置
    print(root.val);
    traverse(root.right);
}
```

那么根据这个性质，我们来做两道算法题。

寻找第 K 小的元素

这是力扣第 230 题「二叉搜索树中第 K 小的元素」，看下题目：

230. 二叉搜索树中第K小的元素 labuladong 题解 思路

难度 中等 302 收藏 分享 切换为英文 接收动态

给定一个二叉搜索树，编写一个函数 `kthSmallest` 来查找其中第 **k** 个最小的元素。

你可以假设 k 总是有效的，1 ≤ k ≤ 二叉搜索树元素个数。

示例:

```
输入: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
输出: 3
```

这个需求很常见吧，一个直接的思路就是升序排序，然后找第 `k` 个元素呗。BST 的中序遍历其实就是升序排序的结果，找第 `k` 个元素肯定不是什么难事。

按照这个思路，可以直接写出代码：

```
int kthSmallest(TreeNode root, int k) {
    // 利用 BST 的中序遍历特性
    traverse(root, k);
    return res;
}

// 记录结果
int res = 0;
// 记录当前元素的排名
int rank = 0;
void traverse(TreeNode root, int k) {
    if (root == null) {
```

```
            return;
        }
        traverse(root.left, k);
        /* 中序遍历代码位置 */
        rank++;
        if (k == rank) {
            // 找到第 k 小的元素
            res = root.val;
            return;
        }
        /*****************/
        traverse(root.right, k);
    }
```

这道题就做完了，不过呢，还是要多说几句，因为这个解法并不是最高效的解法，而是仅仅适用于这道题。

我们旧文 高效计算数据流的中位数 中就提过今天的这个问题：

> 如果让你实现一个在二叉搜索树中通过排名计算对应元素的方法 `select(int k)`，你会怎么设计？

如果按照我们刚才说的方法，利用「BST 中序遍历就是升序排序结果」这个性质，每次寻找第 `k` 小的元素都要中序遍历一次，最坏的时间复杂度是 `O(N)`，`N` 是 BST 的节点个数。

要知道 BST 性质是非常牛逼的，像红黑树这种改良的自平衡 BST，增删查改都是 `O(logN)` 的复杂度，让你算一个第 `k` 小元素，时间复杂度竟然要 `O(N)`，有点低效了。

所以说，计算第 `k` 小元素，最好的算法肯定也是对数级别的复杂度，不过这个依赖于 BST 节点记录的信息有多少。

我们想一下 BST 的操作为什么这么高效？就拿搜索某一个元素来说，BST 能够在对数时间找到该元素的根本原因还是在 BST 的定义里，左子树小右子树大嘛，所以每个节点都可以通过对比自身的值判断去左子树还是右子树搜索目标值，从而避免了全树遍历，达到对数级复杂度。

那么回到这个问题，想找到第 `k` 小的元素，或者说找到排名为 `k` 的元素，如果想达到对数级复杂度，关键也在于每个节点得知道他自己排第几。

比如说你让我查找排名为 `k` 的元素，当前节点知道自己排名第 `m`，那么我可以比较 `m` 和 `k` 的大小：

1、如果 `m == k`，显然就是找到了第 `k` 个元素，返回当前节点就行了。

2、如果 `k < m`，那说明排名第 `k` 的元素在左子树，所以可以去左子树搜索第 `k` 个元素。

3、如果 `k > m`，那说明排名第 `k` 的元素在右子树，所以可以去右子树搜索第 `k - m - 1` 个元素。

这样就可以将时间复杂度降到 `O(logN)` 了。

那么，如何让每一个节点知道自己的排名呢？

这就是我们之前说的，需要在二叉树节点中维护额外信息。**每个节点需要记录，以自己为根的这棵二叉树有多少个节点。**

也就是说，我们 `TreeNode` 中的字段应该如下：

```java
class TreeNode {
    int val;
    // 以该节点为根的树的节点总数
    int size;
    TreeNode left;
    TreeNode right;
}
```

有了 `size` 字段，外加 BST 节点左小右大的性质，对于每个节点 `node` 就可以通过 `node.left` 推导出 `node` 的排名，从而做到我们刚才说到的对数级算法。

当然，`size` 字段需要在增删元素的时候需要被正确维护，力扣提供的 `TreeNode` 是没有 `size` 这个字段的，所以我们这道题就只能利用 BST 中序遍历的特性实现了，但是我们上面说到的优化思路是 BST 的常见操作，还是有必要理解的。

BST 转化累加树

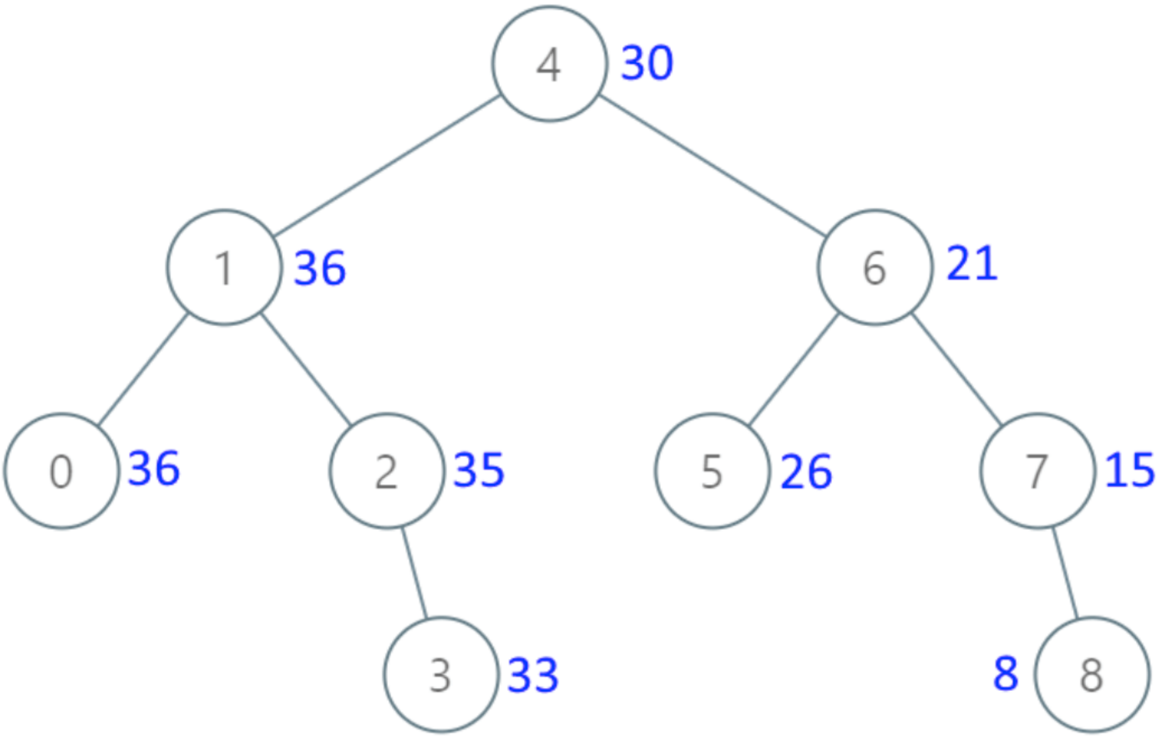
力扣第 538 题和 1038 题都是这道题，完全一样，你可以把它们一块做掉。看下题目：

538. 把二叉搜索树转换为累加树 labuladong 题解 思路

难度 中等 420 收藏 分享 切换为英文 接收动态 反馈

给出二叉 **搜索** 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

示例 1：



题目应该不难理解，比如图中的节点 5，转化成累加树的话，比 5 大的节点有 6，7，8，加上 5 本身，所以累加树上这个节点的值应该是 5+6+7+8=26。

我们需要把 BST 转化成累加树，函数签名如下：

```
TreeNode convertBST(TreeNode root)
```

按照二叉树的通用思路，需要思考每个节点应该做什么，但是这道题上很难想到什么思路。

BST 的每个节点左小右大，这似乎是一个有用的信息，既然累加和是计算大于等于当前值的所有元素之和，那么每个节点都去计算右子树的和，不就行了吗？

这是不行的。对于一个节点来说，确实右子树都是比它大的元素，但问题是它的父节点也可能是比它大的元素呀？这个没法确定的，我们又没有触达父节点的指针，所以二叉树的通用思路在这里用不了。

其实，正确的解法很简单，还是利用 BST 的中序遍历特性。

刚才我们说了 BST 的中序遍历代码可以升序打印节点的值：

```
void traverse(TreeNode root) {
    if (root == null) return;
    traverse(root.left);
    // 中序遍历代码位置
    print(root.val);
    traverse(root.right);
}
```

那如果我想降序打印节点的值怎么办？

很简单，只要把递归顺序改一下就行了：

```
void traverse(TreeNode root) {
    if (root == null) return;
    // 先递归遍历右子树
    traverse(root.right);
    // 中序遍历代码位置
    print(root.val);
    // 后递归遍历左子树
    traverse(root.left);
}
```

这段代码可以降序打印 BST 节点的值，如果维护一个外部累加变量 `sum`，然后把 `sum` 赋值给 BST 中的每一个节点，不就将 BST 转化成累加树了吗？

看下代码就明白了：

```
TreeNode convertBST(TreeNode root) {
    traverse(root);
    return root;
}

// 记录累加和
int sum = 0;
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    traverse(root.right);
    // 维护累加和
    sum += root.val;
    // 将 BST 转化成累加树
    root.val = sum;
    traverse(root.left);
}
```

这道题就解决了，核心还是 BST 的中序遍历特性，只不过我们修改了递归顺序，降序遍历 BST 的元素值，从而契合题目累加树的要求。

简单总结下吧，BST 相关的问题，要么利用 BST 左小右大的特性提升算法效率，要么利用中序遍历的特性满足题目的要求，也就这么些事儿吧。

最后调查下，经过这几篇二叉树相关的系列文章，大家刷题有没有点感觉了？可以留言和我交流。本文对你有帮助的话，请三连~

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



东哥带你刷二叉搜索树（第二期）

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

450. 删除二叉搜索树中的节点（中等）

701. 二叉搜索树中的插入操作（中等）

700. 二叉搜索树中的搜索（简单）

98. 验证二叉搜索树（中等）

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

我们前文 手把手刷二叉搜索树（第一期） 主要是利用二叉搜索树「中序遍历有序」的特性来解决了几道题目，本文来实现 BST 的基础操作：判断 BST 的合法性、增、删、查。其中「删」和「判断合法性」略微复杂。

BST 简介

所谓二叉搜索树（Binary Search Tree，简称 BST）大家应该都不陌生，它是一种特殊的二叉树。

特殊在哪里呢？简单来说就是：左小右大。

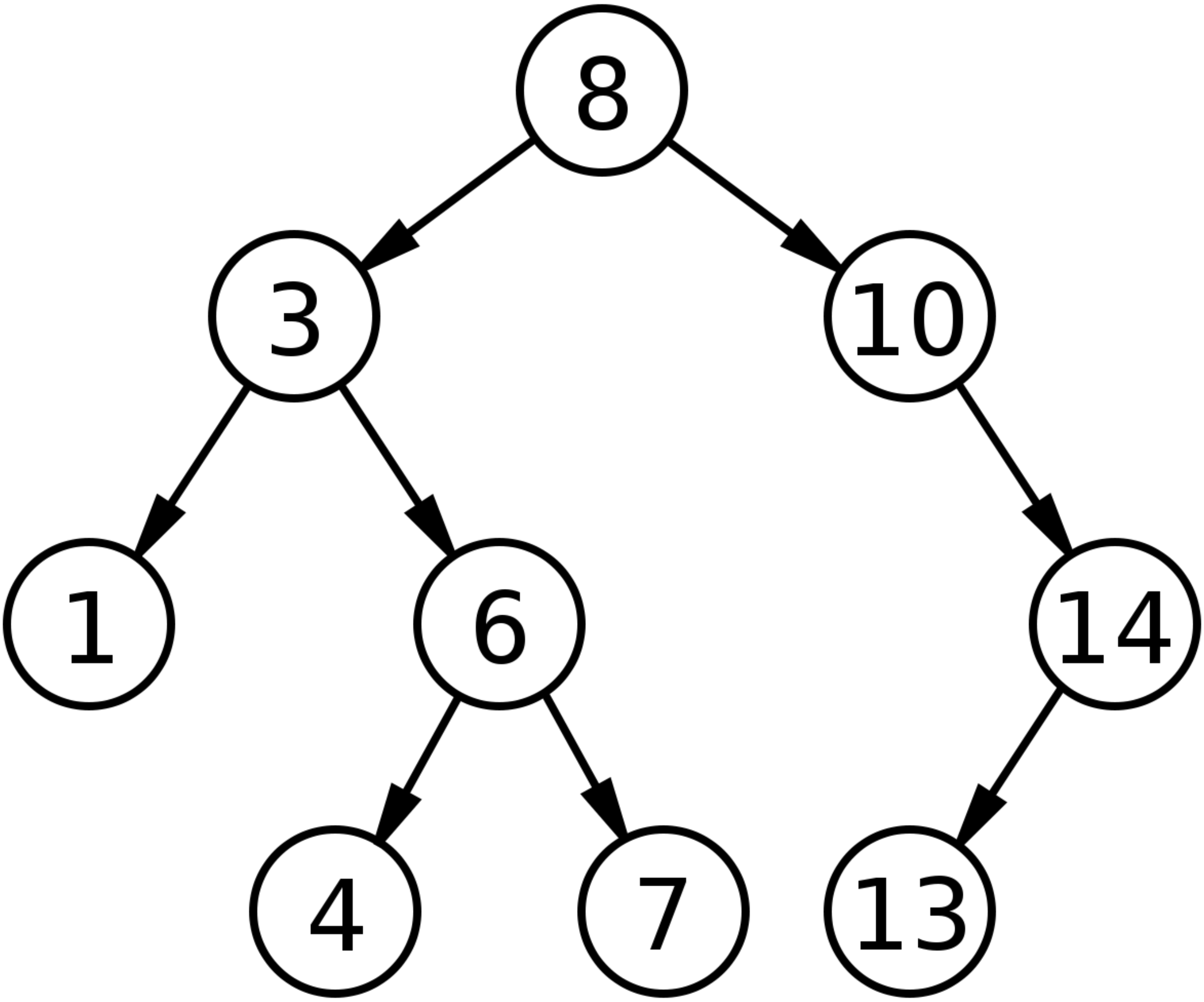
BST 的完整定义如下：

1、BST 中任意一个节点的左子树所有节点的值都小于该节点的值，右子树所有节点的值都大于该节点的值。

2、BST 中任意一个节点的左右子树都是 BST。

有了 BST 的这种特性，就可以在二叉树中做类似二分搜索的操作，搜索一个元素的效率很高。

比如下面这就是一棵合法的二叉树：



对于 BST 相关的问题，你可能会经常看到类似下面这样的代码逻辑：

```
void BST(TreeNode root, int target) {
    if (root.val == target)
        // 找到目标，做点什么
    if (root.val < target)
        BST(root.right, target);
    if (root.val > target)
        BST(root.left, target);
}
```

这个代码框架其实和二叉树的遍历框架差不多，无非就是利用了 BST 左小右大的特性而已。

接下来我们讲几道二叉搜索树的必知必会题目。

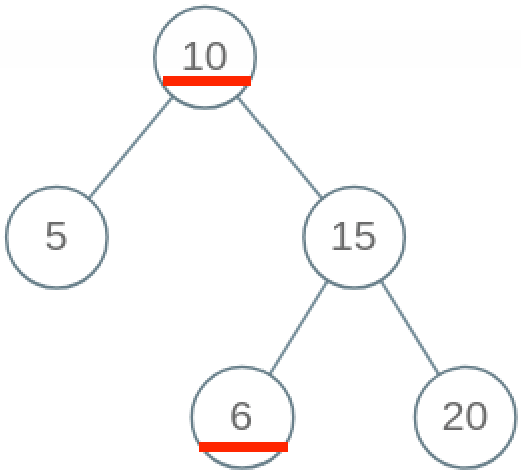
一、判断 BST 的合法性

这里是有坑的哦，我们按照刚才的思路，每个节点自己要做的事不就是比较自己和左右孩子吗？看起来应该这样写代码：

```java
boolean isValidBST(TreeNode root) {
    if (root == null) return true;
    if (root.left != null && root.val <= root.left.val)
        return false;
    if (root.right != null && root.val >= root.right.val)
        return false;

    return isValidBST(root.left)
        && isValidBST(root.right);
}
```

但是这个算法出现了错误，BST 的每个节点应该要小于右边子树的所有节点，下面这个二叉树显然不是 BST，因为节点 10 的右子树中有一个节点 6，但是我们的算法会把它判定为合法 BST：



出现问题的原因在于，对于每一个节点 `root`，代码值检查了它的左右孩子节点是否符合左小右大的原则；但是根据 BST 的定义，`root` 的整个左子树都要小于 `root.val`，整个右子树都要大于 `root.val`。

问题是，对于某一个节点 `root`，他只能管得了自己的左右子节点，怎么把 `root` 的约束传递给左右子树呢？

请看正确的代码：

```java
boolean isValidBST(TreeNode root) {
    return isValidBST(root, null, null);
}

/* 限定以 root 为根的子树节点必须满足 max.val > root.val > min.val */
boolean isValidBST(TreeNode root, TreeNode min, TreeNode max) {
    // base case
    if (root == null) return true;
    // 若 root.val 不符合 max 和 min 的限制，说明不是合法 BST
    if (min != null && root.val <= min.val) return false;
    if (max != null && root.val >= max.val) return false;
    // 限定左子树的最大值是 root.val，右子树的最小值是 root.val
```

```java
        return isValidBST(root.left, min, root)
            && isValidBST(root.right, root, max);
    }
```

我们通过使用辅助函数，增加函数参数列表，在参数中携带额外信息，将这种约束传递给子树的所有节点，这也是二叉树算法的一个小技巧吧。

在 BST 中搜索元素

力扣第 700 题「二叉搜索树中的搜索」就是让你在 BST 中搜索值为 `target` 的节点，函数签名如下：

```java
TreeNode searchBST(TreeNode root, int target);
```

如果是在一棵普通的二叉树中寻找，可以这样写代码：

```java
TreeNode searchBST(TreeNode root, int target);
    if (root == null) return null;
    if (root.val == target) return root;
    // 当前节点没找到就递归地去左右子树寻找
    TreeNode left = searchBST(root.left, target);
    TreeNode right = searchBST(root.right, target);

    return left != null ? left : right;
}
```

这样写完全正确，但这段代码相当于穷举了所有节点，适用于所有普通二叉树。那么应该如何充分利用信息，把 BST 这个「左小右大」的特性用上？

很简单，其实不需要递归地搜索两边，类似二分查找思想，根据 `target` 和 `root.val` 的大小比较，就能排除一边。我们把上面的思路稍稍改动：

```java
TreeNode searchBST(TreeNode root, int target) {
    if (root == null) {
        return null;
    }
    // 去左子树搜索
    if (root.val > target) {
        return searchBST(root.left, target);
    }
    // 去右子树搜索
    if (root.val < target) {
        return searchBST(root.right, target);
    }
    return root;
}
```

在 BST 中插入一个数

对数据结构的操作无非遍历 + 访问，遍历就是「找」，访问就是「改」。具体到这个问题，插入一个数，就是先找到插入位置，然后进行插入操作。

上一个问题，我们总结了 BST 中的遍历框架，就是「找」的问题。直接套框架，加上「改」的操作即可。**一旦涉及「改」，函数就要返回 `TreeNode` 类型，并且对递归调用的返回值进行接收。**

```
TreeNode insertIntoBST(TreeNode root, int val) {
    // 找到空位置插入新节点
    if (root == null) return new TreeNode(val);
    // if (root.val == val)
    //     BST 中一般不会插入已存在元素
    if (root.val < val)
        root.right = insertIntoBST(root.right, val);
    if (root.val > val)
        root.left = insertIntoBST(root.left, val);
    return root;
}
```

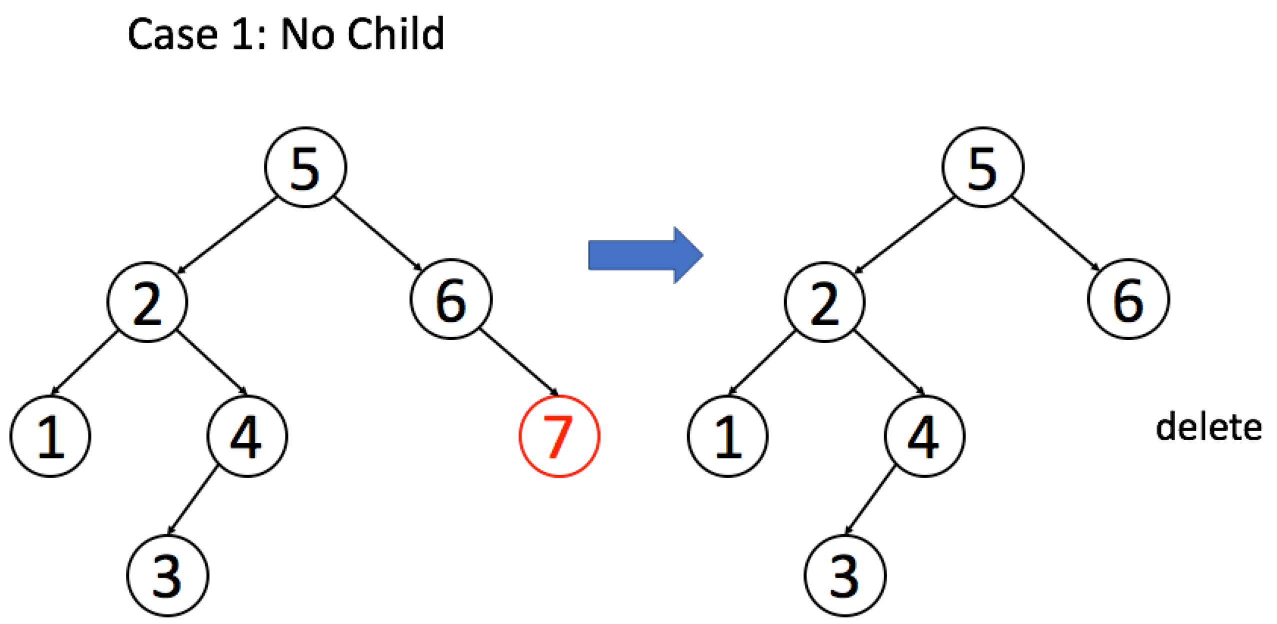
三、在 BST 中删除一个数

这个问题稍微复杂，跟插入操作类似，先「找」再「改」，先把框架写出来再说：

```
TreeNode deleteNode(TreeNode root, int key) {
    if (root.val == key) {
        // 找到啦，进行删除
    } else if (root.val > key) {
        // 去左子树找
        root.left = deleteNode(root.left, key);
    } else if (root.val < key) {
        // 去右子树找
        root.right = deleteNode(root.right, key);
    }
    return root;
}
```

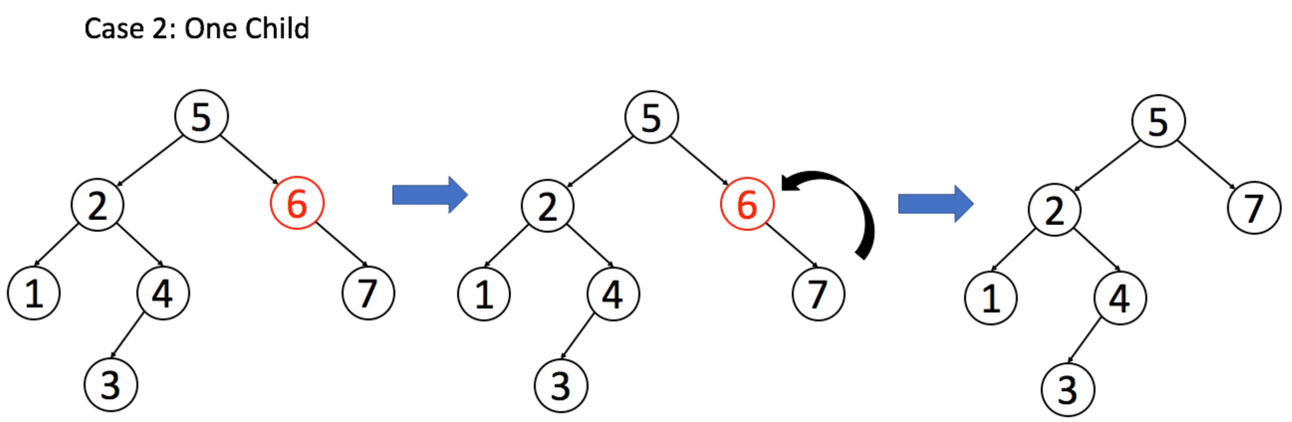
找到目标节点了，比方说是节点 `A`，如何删除这个节点，这是难点。因为删除节点的同时不能破坏 BST 的性质。有三种情况，用图片来说明。

情况 1：`A` 恰好是末端节点，两个子节点都为空，那么它可以当场去世了。



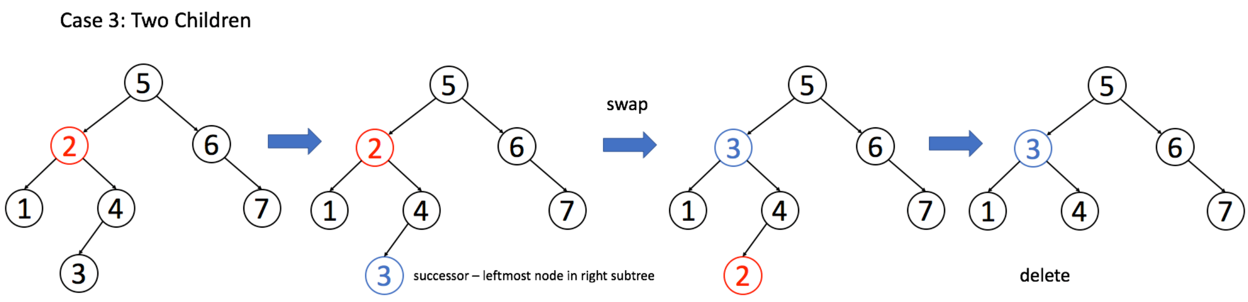

```
if (root.left == null && root.right == null)
    return null;
```

情况 2：A 只有一个非空子节点，那么它要让这个孩子接替自己的位置。




```
// 排除了情况 1 之后
if (root.left == null) return root.right;
if (root.right == null) return root.left;
```

情况 3：A 有两个子节点，麻烦了，为了不破坏 BST 的性质，A 必须找到左子树中最大的那个节点，或者右子树中最小的那个节点来接替自己。我们以第二种方式讲解。




```
if (root.left != null && root.right != null) {
    // 找到右子树的最小节点
    TreeNode minNode = getMin(root.right);
    // 把 root 改成 minNode
    root.val = minNode.val;
    // 转而去删除 minNode
    root.right = deleteNode(root.right, minNode.val);
}
```

三种情况分析完毕，填入框架，简化一下代码：

```
TreeNode deleteNode(TreeNode root, int key) {
    if (root == null) return null;
    if (root.val == key) {
        // 这两个 if 把情况 1 和 2 都正确处理了
        if (root.left == null) return root.right;
        if (root.right == null) return root.left;
        // 处理情况 3
        // 获得右子树最小的节点
        TreeNode minNode = getMin(root.right);
        // 删除右子树最小的节点
        root.right = deleteNode(root.right, minNode.val);
        // 用右子树最小的节点替换 root 节点
        minNode.left = root.left;
        minNode.right = root.right;
        root = minNode;
    } else if (root.val > key) {
        root.left = deleteNode(root.left, key);
    } else if (root.val < key) {
        root.right = deleteNode(root.right, key);
    }
    return root;
}

TreeNode getMin(TreeNode node) {
    // BST 最左边的就是最小的
    while (node.left != null) node = node.left;
    return node;
}
```

这样，删除操作就完成了。

注意一下，上述代码在处理情况 3 时通过一系列略微复杂的链表操作交换 root 和 minNode 两个节点：

```
// 处理情况 3
// 获得右子树最小的节点
TreeNode minNode = getMin(root.right);
// 删除右子树最小的节点
root.right = deleteNode(root.right, minNode.val);
// 用右子树最小的节点替换 root 节点
minNode.left = root.left;
minNode.right = root.right;
root = minNode;
```

有的读者可能会疑惑，替换 root 节点为什么这么麻烦，直接改 val 字段不就行了？看起来还更简洁易懂：

```
// 处理情况 3
// 获得右子树最小的节点
TreeNode minNode = getMin(root.right);
// 删除右子树最小的节点
root.right = deleteNode(root.right, minNode.val);
// 用右子树最小的节点替换 root 节点
root.val = minNode.val;
```

仅对于这道算法题来说是可以的，但这样操作并不完美，我们一般不会通过修改节点内部的值来交换节点。

因为在实际应用中，BST 节点内部的数据域是用户自定义的，可以非常复杂，而 BST 作为数据结构（一个工具人），其操作应该和内部存储的数据域解耦，所以我们更倾向于使用指针操作来交换节点，根本没必要关心内部数据。

不过这里我们暂时忽略这个细节，旨在突出 BST 基本操作的共性，以及借助框架逐层细化问题的思维方式。

最后总结

通过这篇文章，我们总结出了如下几个技巧：

1、如果当前节点会对下面的子节点有整体影响，可以通过辅助函数增长参数列表，借助参数传递信息。

2、在二叉树递归框架之上，扩展出一套 BST 代码框架：

```
void BST(TreeNode root, int target) {
    if (root.val == target)
        // 找到目标，做点什么
    if (root.val < target)
        BST(root.right, target);
    if (root.val > target)
        BST(root.left, target);
}
```

3、根据代码框架掌握了 BST 的增删查改操作。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



东哥带你刷二叉搜索树（第三期）

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

96. 不同的二叉搜索树（简单）

95. 不同的二叉搜索树II（中等）

PS：刷题插件 集成了手把手刷二叉树功能，按照公式和套路讲解了 150 道二叉树题目，可手把手带你刷完二叉树分类的题目，迅速掌握递归思维。

之前写了两篇手把手刷 BST 算法题的文章，第一篇 讲了中序遍历对 BST 的重要意义，第二篇 写了 BST 的基本操作。

本文就来写手把手刷 BST 系列的第三篇，循序渐进地讲两道题，如何计算所有有效 BST。

第一道题是力扣第 96 题「不同的二叉搜索树」，给你输入一个正整数 `n`，请你计算，存储 `{1,2,3...,n}` 这些值共有多少种不同的 BST 结构。

函数签名如下：

```
int numTrees(int n);
```

比如说输入 `n = 3`，算法返回 5，因为共有如下 5 种不同的 BST 结构存储 `{1,2,3}`：

```
n = 3 时有如下 5 种不同的 BST 结果:

   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

这就是一个正宗的穷举问题，那么什么方式能够正确地穷举有效 BST 的数量呢?

我们前文说过，不要小看「穷举」，这是一件看起来简单但是比较有技术含量的事情，问题的关键就是不能数漏，也不能数多，你咋整?

———————————————

应合作方要求，本文不便在此发布，请扫码关注回复关键词「BST」查看:

2.3 图论

图论在实际笔试中考的不多，但它的经典算法比较多，比如什么最小生成树，最短路径，拓扑排序，二分图判定之类的。

所以本章围绕图论的经典算法展开，太难的图论算法我觉得咱是没多大必要掌握的。

公众号标签：图论算法

图论基础

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong



学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

797. 所有可能的路径（中等）

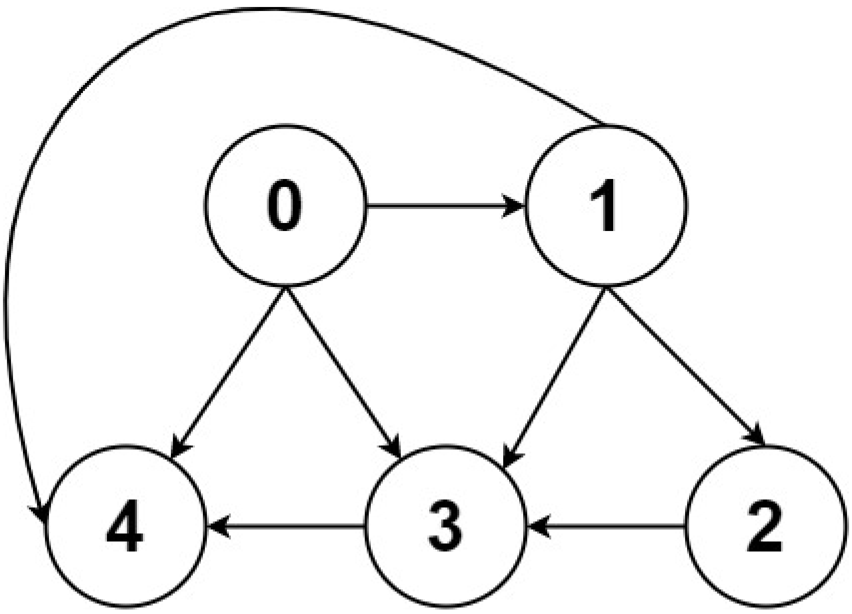
经常有读者问我「图」这种数据结构，其实我在 学习数据结构和算法的框架思维 中说过，虽然图可以玩出更多的算法，解决更复杂的问题，但本质上图可以认为是多叉树的延伸。

面试笔试很少出现图相关的问题，就算有，大多也是简单的遍历问题，基本上可以完全照搬多叉树的遍历。

那么，本文依然秉持我们号的风格，只讲「图」最实用的，离我们最近的部分，让你心里对图有个直观的认识，文末我给出了其他经典图论算法，理解本文后应该都可以拿下的。

图的逻辑结构和具体实现

一幅图是由**节点**和**边**构成的，逻辑结构如下：



什么叫「逻辑结构」？就是说为了方便研究，我们把图抽象成这个样子。

根据这个逻辑结构，我们可以认为每个节点的实现如下：

```
/* 图节点的逻辑结构 */
class Vertex {
    int id;
    Vertex[] neighbors;
}
```

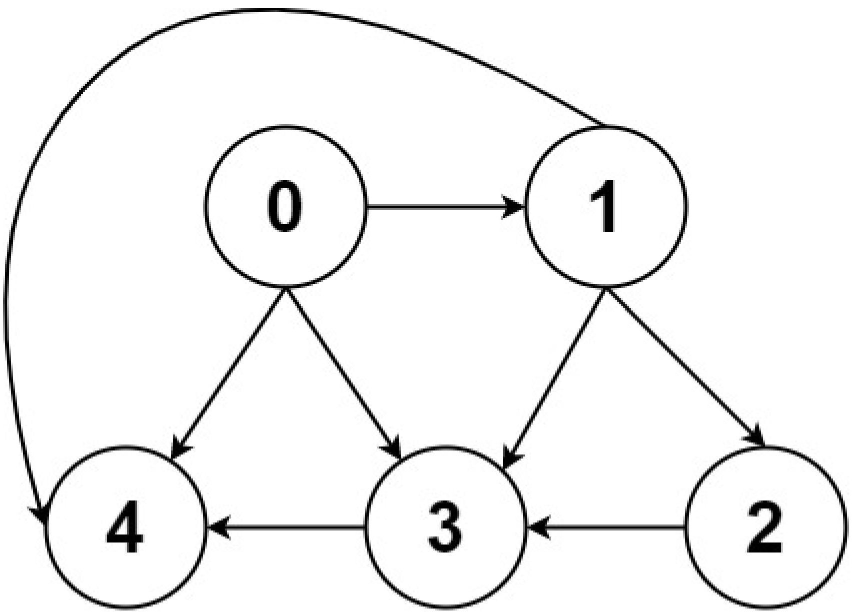
看到这个实现，你有没有很熟悉？它和我们之前说的多叉树节点几乎完全一样：

```
/* 基本的 N 叉树节点 */
class TreeNode {
    int val;
    TreeNode[] children;
}
```

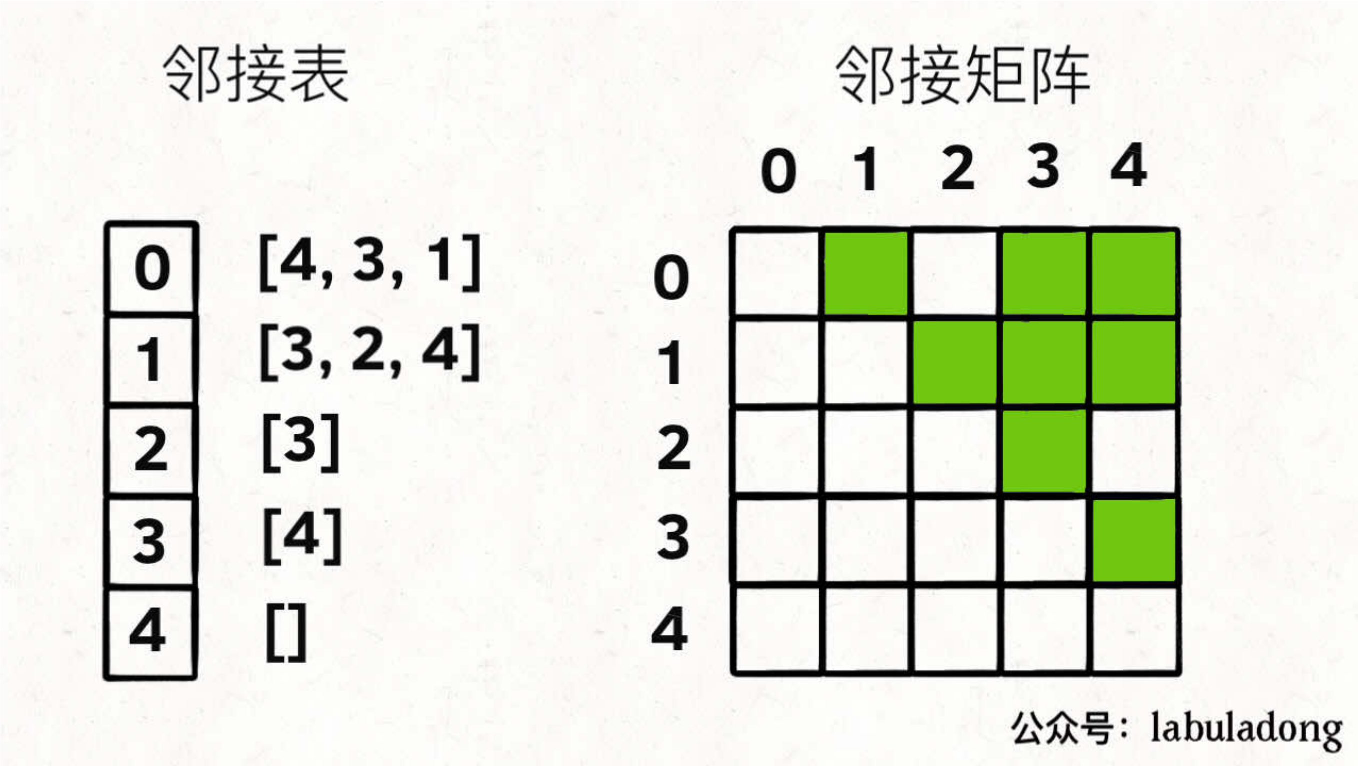
所以说，图真的没啥高深的，本质上就是个高级点的多叉树而已，适用于树的 DFS/BFS 遍历算法，全部适用于图。

不过呢，上面的这种实现是「逻辑上的」，实际上我们很少用这个 `Vertex` 类实现图，而是用常说的**邻接表和邻接矩阵**来实现。

比如还是刚才那幅图：



用邻接表和邻接矩阵的存储方式如下：



邻接表很直观，我把每个节点 x 的邻居都存到一个列表里，然后把 x 和这个列表关联起来，这样就可以通过一个节点 x 找到它的所有相邻节点。

邻接矩阵则是一个二维布尔数组，我们权且称为 matrix，如果节点 x 和 y 是相连的，那么就把 matrix[x][y] 设为 true（上图中绿色的方格代表 true）。如果想找节点 x 的邻居，去扫一圈 matrix[x][..] 就行了。

如果用代码的形式来表现，邻接表和邻接矩阵大概长这样：

```
// 邻接表
// graph[x] 存储 x 的所有邻居节点
List<Integer>[] graph;

// 邻接矩阵
// matrix[x][y] 记录 x 是否有一条指向 y 的边
boolean[][] matrix;
```

那么，为什么有这两种存储图的方式呢？肯定是因为他们各有优劣。

对于邻接表，好处是占用的空间少。

你看邻接矩阵里面空着那么多位置，肯定需要更多的存储空间。

但是，邻接表无法快速判断两个节点是否相邻。

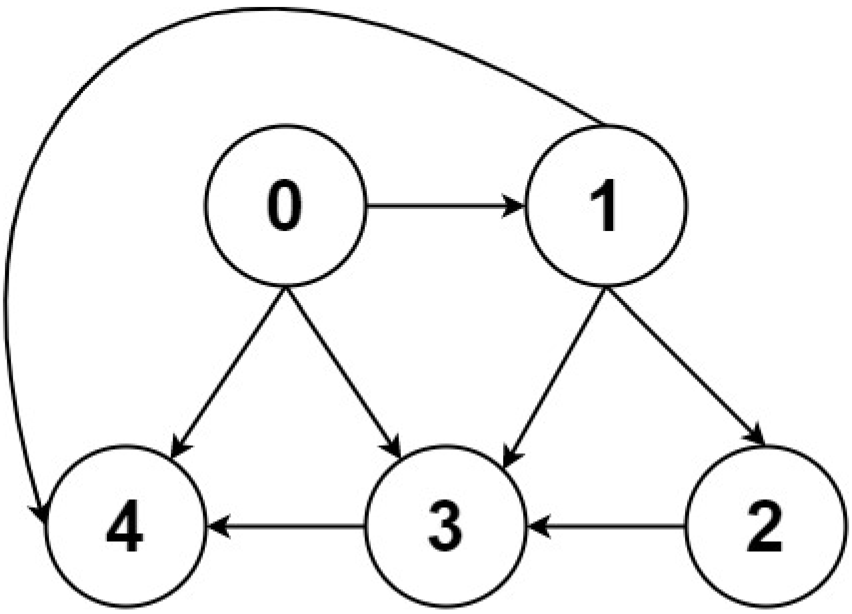
比如说我想判断节点 1 是否和节点 3 相邻，我要去邻接表里 1 对应的邻居列表里查找 3 是否存在。但对于邻接矩阵就简单了，只要看看 matrix[1][3] 就知道了，效率高。

所以说，使用哪一种方式实现图，要看具体情况。

> PS：在常规的算法题中，邻接表的使用会更频繁一些，主要是因为操作起来较为简单，但这不意味着邻接矩阵应该被轻视。矩阵是一个强有力的数学工具，图的一些隐晦性质可以借助精妙的矩阵运算展现出来。不过本文不准备引入数学内容，所以有兴趣的读者可以自行搜索学习。

最后，我们再明确一个图论中特有的**度**（degree）的概念，在无向图中，「度」就是每个节点相连的边的条数。

由于有向图的边有方向，所以有向图中每个节点「度」被细分为**入度**（indegree）和**出度**（outdegree），比如下图：



其中节点 `3` 的入度为 3（有三条边指向它），出度为 1（它有 1 条边指向别的节点）。

好了，对于「图」这种数据结构，能看懂上面这些就绰绰够用了。

那你可能会问，我们上面说的这个图的模型仅仅是「有向无权图」，不是还有什么加权图，无向图，等等……

其实，这些更复杂的模型都是基于这个最简单的图衍生出来的。

有向加权图怎么实现？很简单呀：

如果是邻接表，我们不仅仅存储某个节点 `x` 的所有邻居节点，还存储 `x` 到每个邻居的权重，不就实现加权有向图了吗？

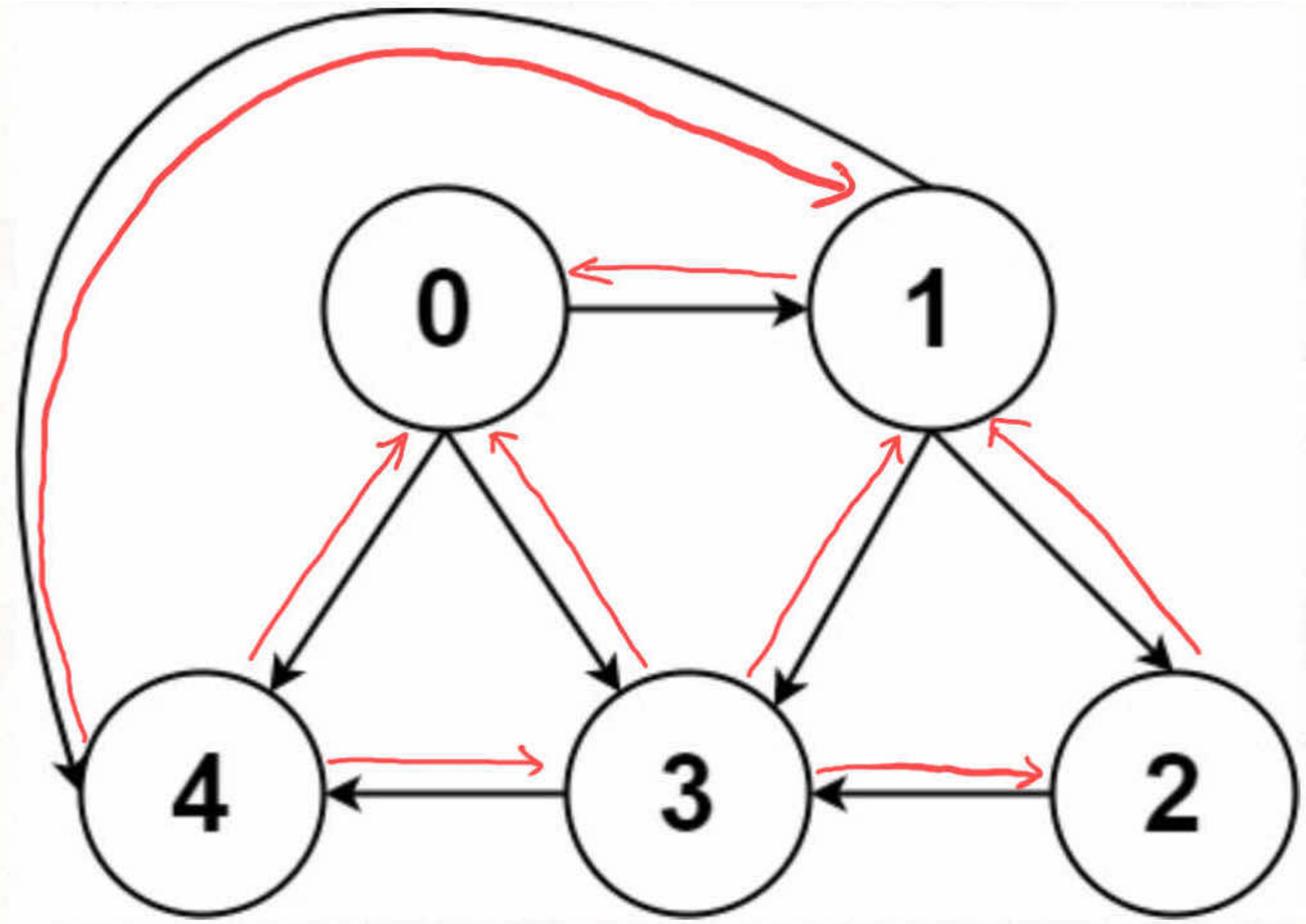
如果是邻接矩阵，`matrix[x][y]` 不再是布尔值，而是一个 int 值，0 表示没有连接，其他值表示权重，不就变成加权有向图了吗？

如果用代码的形式来表现，大概长这样：

```
// 邻接表
// graph[x] 存储 x 的所有邻居节点以及对应的权重
List<int[]>[] graph;

// 邻接矩阵
// matrix[x][y] 记录 x 指向 y 的边的权重，0 表示不相邻
int[][] matrix;
```

无向图怎么实现？也很简单，所谓的「无向」，是不是等同于「双向」？



如果连接无向图中的节点 x 和 y，把 matrix[x][y] 和 matrix[y][x] 都变成 true 不就行了；邻接表也是类似的操作，在 x 的邻居列表里添加 y，同时在 y 的邻居列表里添加 x。

把上面的技巧合起来，就变成了无向加权图……

好了，关于图的基本介绍就到这里，现在不管来什么乱七八糟的图，你心里应该都有底了。

下面来看看所有数据结构都逃不过的问题：遍历。

图的遍历

学习数据结构和算法的框架思维 说过，各种数据结构被发明出来无非就是为了遍历和访问，所以「遍历」是所有数据结构的基础。

图怎么遍历？还是那句话，参考多叉树，多叉树的遍历框架如下：

```
/* 多叉树遍历框架 */
void traverse(TreeNode root) {
    if (root == null) return;

    for (TreeNode child : root.children) {
        traverse(child);
```

```
    }
}
```

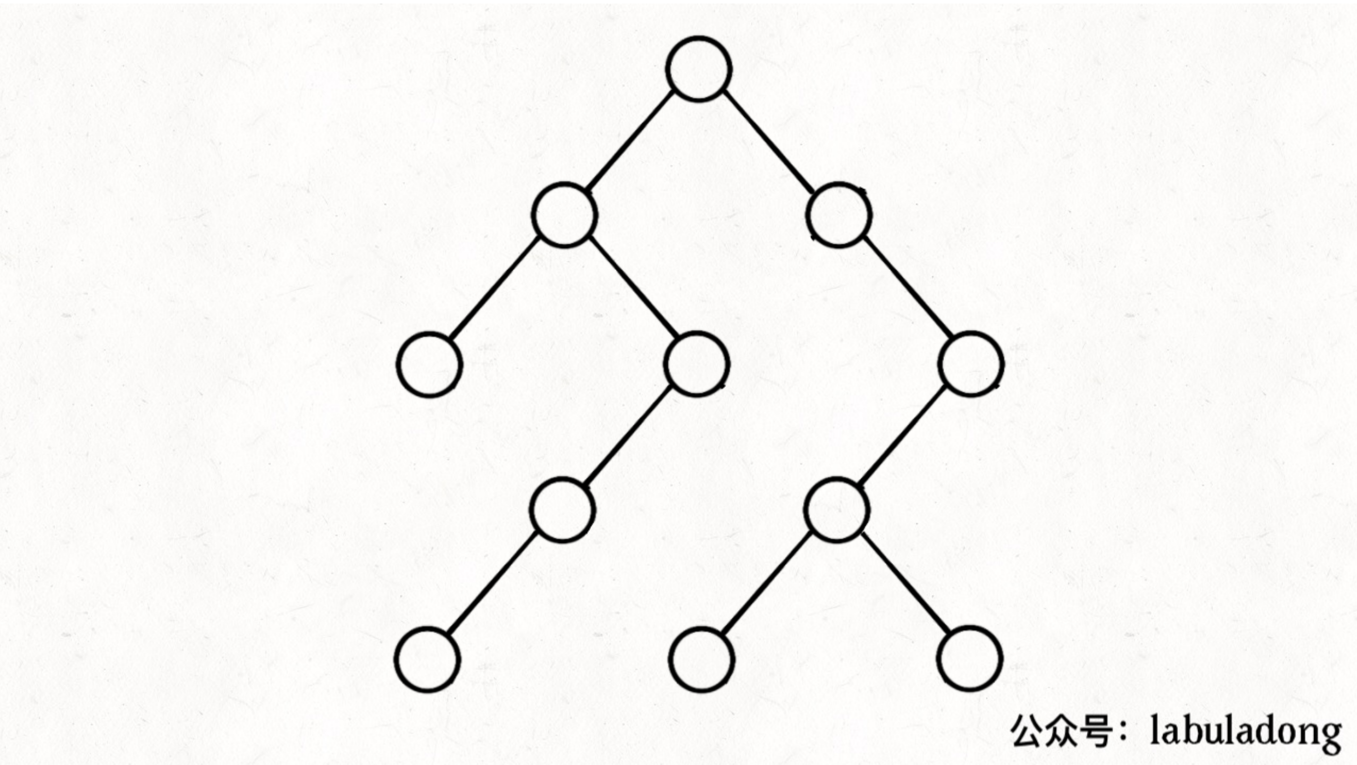
图和多叉树最大的区别是，图是可能包含环的，你从图的某一个节点开始遍历，有可能走了一圈又回到这个节点。

所以，如果图包含环，遍历框架就要一个 `visited` 数组进行辅助：

```java
// 记录被遍历过的节点
boolean[] visited;
// 记录从起点到当前节点的路径
boolean[] onPath;

/* 图遍历框架 */
void traverse(Graph graph, int s) {
    if (visited[s]) return;
    // 经过节点 s，标记为已遍历
    visited[s] = true;
    // 做选择：标记节点 s 在路径上
    onPath[s] = true;
    for (int neighbor : graph.neighbors(s)) {
        traverse(graph, neighbor);
    }
    // 撤销选择：节点 s 离开路径
    onPath[s] = false;
}
```

注意 `visited` 数组和 `onPath` 数组的区别，因为二叉树算是特殊的图，所以用遍历二叉树的过程来理解下这两个数组的区别：



上述 GIF 描述了递归遍历二叉树的过程，在 `visited` 中被标记为 true 的节点用灰色表示，在 `onPath` 中被标记为 true 的节点用绿色表示，这下你可以理解它们二者的区别了吧。

如果让你处理路径相关的问题，这个 `onPath` 变量是肯定会被用到的，比如 拓扑排序 中就有运用。

另外，你应该注意到了，这个 `onPath` 数组的操作很像 回溯算法核心套路 中做「做选择」和「撤销选择」，区别在于位置：回溯算法的「做选择」和「撤销选择」在 for 循环里面，而对 `onPath` 数组的操作在 for 循环外面。

在 for 循环里面和外面唯一的区别就是对根节点的处理。

比如下面两种多叉树的遍历：

```
void traverse(TreeNode root) {
    if (root == null) return;
    System.out.println("enter: " + root.val);
    for (TreeNode child : root.children) {
        traverse(child);
    }
    System.out.println("leave: " + root.val);
}

void traverse(TreeNode root) {
    if (root == null) return;
    for (TreeNode child : root.children) {
        System.out.println("enter: " + child.val);
        traverse(child);
        System.out.println("leave: " + child.val);
    }
}
```

前者会正确打印所有节点的进入和离开信息，而后者唯独会少打印整棵树根节点的进入和离开信息。

为什么回溯算法框架会用后者？因为回溯算法关注的不是节点，而是树枝，不信你看 回溯算法核心套路 里面的图。

显然，对于这里「图」的遍历，我们应该把 `onPath` 的操作放到 for 循环外面，否则会漏掉记录起始点的遍历。

说了这么多 `onPath` 数组，再说下 `visited` 数组，其目的很明显了，由于图可能含有环，`visited` 数组就是防止递归重复遍历同一个节点进入死循环的。

当然，如果题目告诉你图中不含环，可以把 `visited` 数组都省掉，基本就是多叉树的遍历。

题目实践

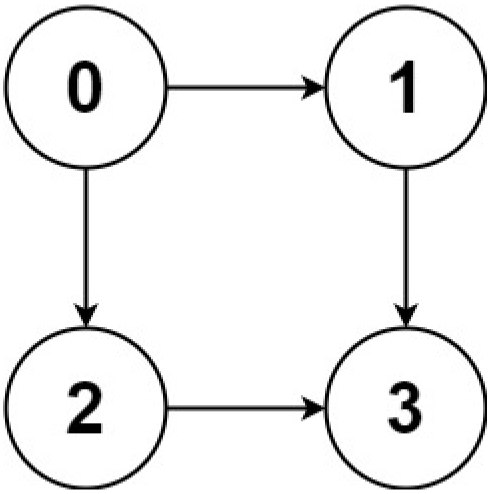
下面我们来看力扣第 797 题「所有可能路径」，函数签名如下：

```
List<List<Integer>> allPathsSourceTarget(int[][] graph);
```

题目输入一幅**有向无环图**，这个图包含 `n` 个节点，标号为 `0, 1, 2,..., n - 1`，请你计算所有从节点 `0` 到节点 `n - 1` 的路径。

输入的这个 `graph` 其实就是「邻接表」表示的一幅图，`graph[i]` 存储这节点 `i` 的所有邻居节点。

比如输入 `graph = [[1,2],[3],[3],[]]`，就代表下面这幅图：



算法应该返回 `[[0,1,3],[0,2,3]]`，即 `0` 到 `3` 的所有路径。

解法很简单，以 `0` 为起点遍历图，同时记录遍历过的路径，当遍历到终点时将路径记录下来即可。

既然输入的图是无环的，我们就不需要 `visited` 数组辅助了，直接套用图的遍历框架：

```
// 记录所有路径
List<List<Integer>> res = new LinkedList<>();

public List<List<Integer>> allPathsSourceTarget(int[][] graph) {
    // 维护递归过程中经过的路径
    LinkedList<Integer> path = new LinkedList<>();
    traverse(graph, 0, path);
    return res;
}

/* 图的遍历框架 */
void traverse(int[][] graph, int s, LinkedList<Integer> path) {

    // 添加节点 s 到路径
    path.addLast(s);

    int n = graph.length;
    if (s == n - 1) {
        // 到达终点
        res.add(new LinkedList<>(path));
        path.removeLast();
        return;
    }

    // 递归每个相邻节点
    for (int v : graph[s]) {
```

```
            traverse(graph, v, path);
        }

        // 从路径移出节点 s
        path.removeLast();
    }
```

这道题就这样解决了，注意 Java 的语言特性，向 `res` 中添加 `path` 时需要拷贝一个新的列表，否则最终 `res` 中的列表都是空的。

最后总结一下，图的存储方式主要有邻接表和邻接矩阵，无论什么花里胡哨的图，都可以用这两种方式存储。

在笔试中，最常考的算法是图的遍历，和多叉树的遍历框架是非常类似的。

当然，图还会有很多其他的有趣算法，比如 二分图判定，环检测和拓扑排序（编译器循环引用检测就是类似的算法），最小生成树，Dijkstra 最短路径算法 等等，有兴趣的读者可以去看看，本文就到这了。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



拓扑排序

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

207. 课程表

210. 课程表 II

图这种数据结构有一些比较特殊的算法，比如二分图判断，有环图无环图的判断，拓扑排序，以及最经典的最小生成树，单源最短路径问题，更难的就是类似网络流这样的问题。

不过以我的经验呢，像网络流这种问题，你又不是打竞赛的，没时间的话就没必要学了；像 最小生成树 和 最短路径问题，虽然从刷题的角度用到的不多，但它们属于经典算法，学有余力可以掌握一下；像 二分图判定、拓扑排序这一类，属于比较基本且有用的算法，应该比较熟练地掌握。

那么本文就结合具体的算法题，来说两个图论算法：有向图的环检测、拓扑排序算法。

这两个算法既可以用 DFS 思路解决，也可以用 BFS 思路解决，相对而言 BFS 解法从代码实现上看更简洁一些，但 DFS 解法有助于你进一步理解递归遍历数据结构的奥义，所以本文中我先讲 DFS 遍历的思路，再讲 BFS 遍历的思路。

环检测算法（DFS 版本）

先来看看力扣第 207 题「课程表」：

207. 课程表 labuladong 题解 思路

难度 中等 896

你这个学期必须选修 `numCourses` 门课程，记为 `0` 到 `numCourses - 1` 。

在选修某些课程之前需要一些先修课程。 先修课程按数组 `prerequisites` 给出，其中 $prerequisites[i] = [a_i, b_i]$ ，表示如果要学习课程 a_i 则 **必须** 先学习课程 b_i 。

- 例如，先修课程对 `[0, 1]` 表示：想要学习课程 `0` ，你需要先完成课程 `1` 。

请你判断是否可能完成所有课程的学习？如果可以，返回 `true` ；否则，返回 `false` 。

示例 1：

```
输入：numCourses = 2, prerequisites = [[1,0]]
输出：true
解释：总共有 2 门课程。学习课程 1 之前，你需要完成课程 0 。这是可能的。
```

示例 2：

```
输入：numCourses = 2, prerequisites = [[1,0],[0,1]]
输出：false
解释：总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0 ；并且学习课程 0
之前，你还应先完成课程 1 。这是不可能的。
```

函数签名如下：

```java
boolean canFinish(int numCourses, int[][] prerequisites);
```

题目应该不难理解，什么时候无法修完所有课程？当存在循环依赖的时候。

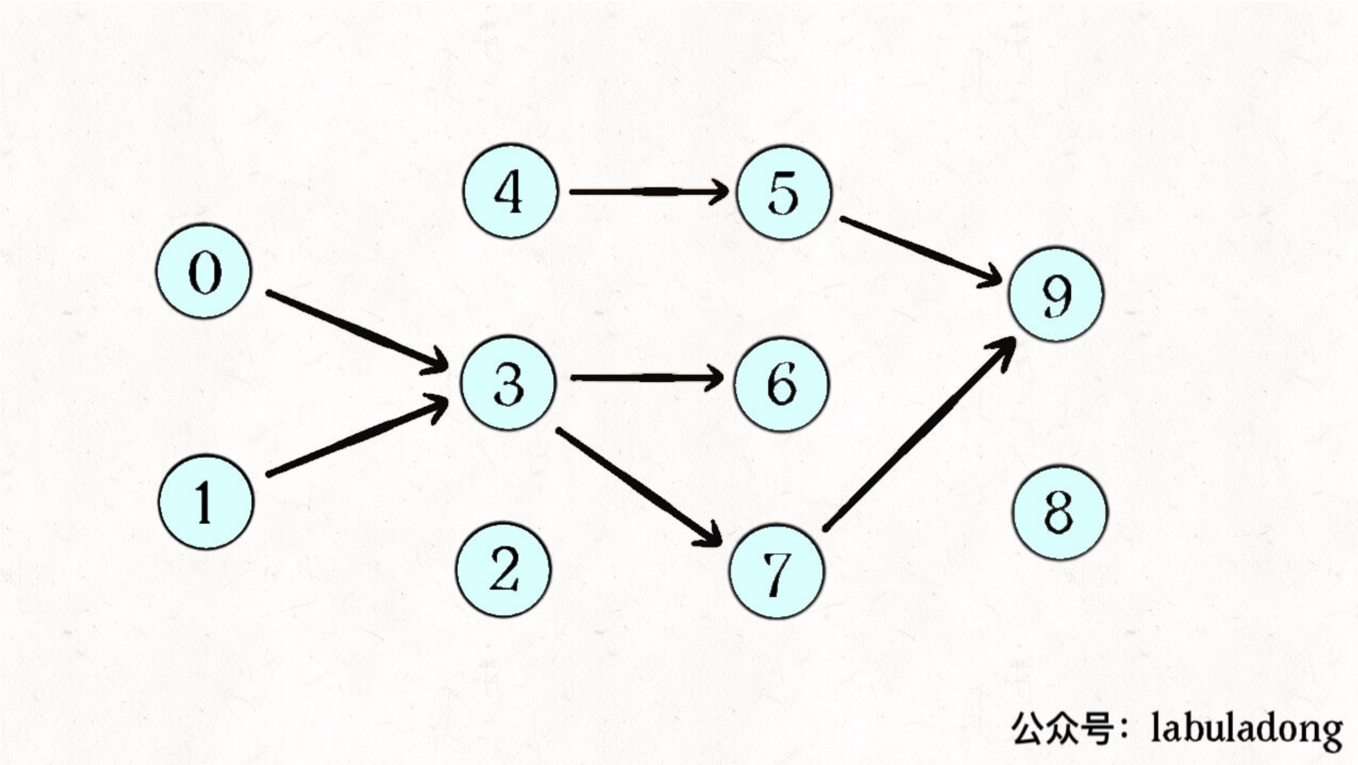
其实这种场景在现实生活中也十分常见，比如我们写代码 import 包也是一个例子，必须合理设计代码目录结构，否则会出现循环依赖，编译器会报错，所以编译器实际上也使用了类似算法来判断你的代码是否能够成功编译。

看到依赖问题，首先想到的就是把问题转化成「有向图」这种数据结构，只要图中存在环，那就说明存在循环依赖。

具体来说，我们首先可以把课程看成「有向图」中的节点，节点编号分别是 `0, 1, ..., numCourses-1`，把课程之间的依赖关系看做节点之间的有向边。

比如说必须修完课程 1 才能去修课程 3，那么就有一条有向边从节点 1 指向 3。

所以我们可以根据题目输入的 prerequisites 数组生成一幅类似这样的图：



如果发现这幅有向图中存在环，那就说明课程之间存在循环依赖，肯定没办法全部上完；反之，如果没有环，那么肯定能上完全部课程。

好，那么想解决这个问题，首先我们要把题目的输入转化成一幅有向图，然后再判断图中是否存在环。

如何转换成图呢？我们前文 图论基础 写过图的两种存储形式，邻接矩阵和邻接表。

以我刷题的经验，常见的存储方式是使用邻接表，比如下面这种结构：

```
List<Integer>[] graph;
```

graph[s] 是一个列表，存储着节点 s 所指向的节点。

所以我们首先可以写一个建图函数：

```
List<Integer>[] buildGraph(int numCourses, int[][] prerequisites) {
    // 图中共有 numCourses 个节点
    List<Integer>[] graph = new LinkedList[numCourses];
    for (int i = 0; i < numCourses; i++) {
        graph[i] = new LinkedList<>();
    }
    for (int[] edge : prerequisites) {
        int from = edge[1], to = edge[0];
        // 添加一条从 from 指向 to 的有向边
        // 边的方向是「被依赖」关系，即修完课程 from 才能修课程 to
        graph[from].add(to);
```

```
    }
    return graph;
}
```

图建出来了，怎么判断图中有没有环呢？

先不要急，我们先来思考如何遍历这幅图，只要会遍历，就可以判断图中是否存在环了。

前文 图论基础 写了 DFS 算法遍历图的框架，无非就是从多叉树遍历框架扩展出来的，加了个 `visited` 数组罢了：

```
// 防止重复遍历同一个节点
boolean[] visited;
// 从节点 s 开始 DFS 遍历，将遍历过的节点标记为 true
void traverse(List<Integer>[] graph, int s) {
    if (visited[s]) {
        return;
    }
    /* 前序遍历代码位置 */
    // 将当前节点标记为已遍历
    visited[s] = true;
    for (int t : graph[s]) {
        traverse(graph, t);
    }
    /* 后序遍历代码位置 */
}
```

那么我们就可以直接套用这个遍历代码：

```
// 防止重复遍历同一个节点
boolean[] visited;

boolean canFinish(int numCourses, int[][] prerequisites) {
    List<Integer>[] graph = buildGraph(numCourses, prerequisites);

    visited = new boolean[numCourses];
    for (int i = 0; i < numCourses; i++) {
        traverse(graph, i);
    }
}

void traverse(List<Integer>[] graph, int s) {
    // 代码见上文
}
```

注意图中并不是所有节点都相连，所以要用一个 for 循环将所有节点都作为起点调用一次 DFS 搜索算法。

这样，就能遍历这幅图中的所有节点了，你打印一下 `visited` 数组，应该全是 true。

前文 学习数据结构和算法的框架思维 说过，图的遍历和遍历多叉树差不多，所以到这里你应该都能很容易理解。

现在可以思考如何判断这幅图中是否存在环。

我们前文 回溯算法核心套路详解 说过，你可以把递归函数看成一个在递归树上游走的指针，这里也是类似的：

你也可以把 `traverse` 看做在图中节点上游走的指针，只需要再添加一个布尔数组 `onPath` 记录当前 `traverse` 经过的路径：

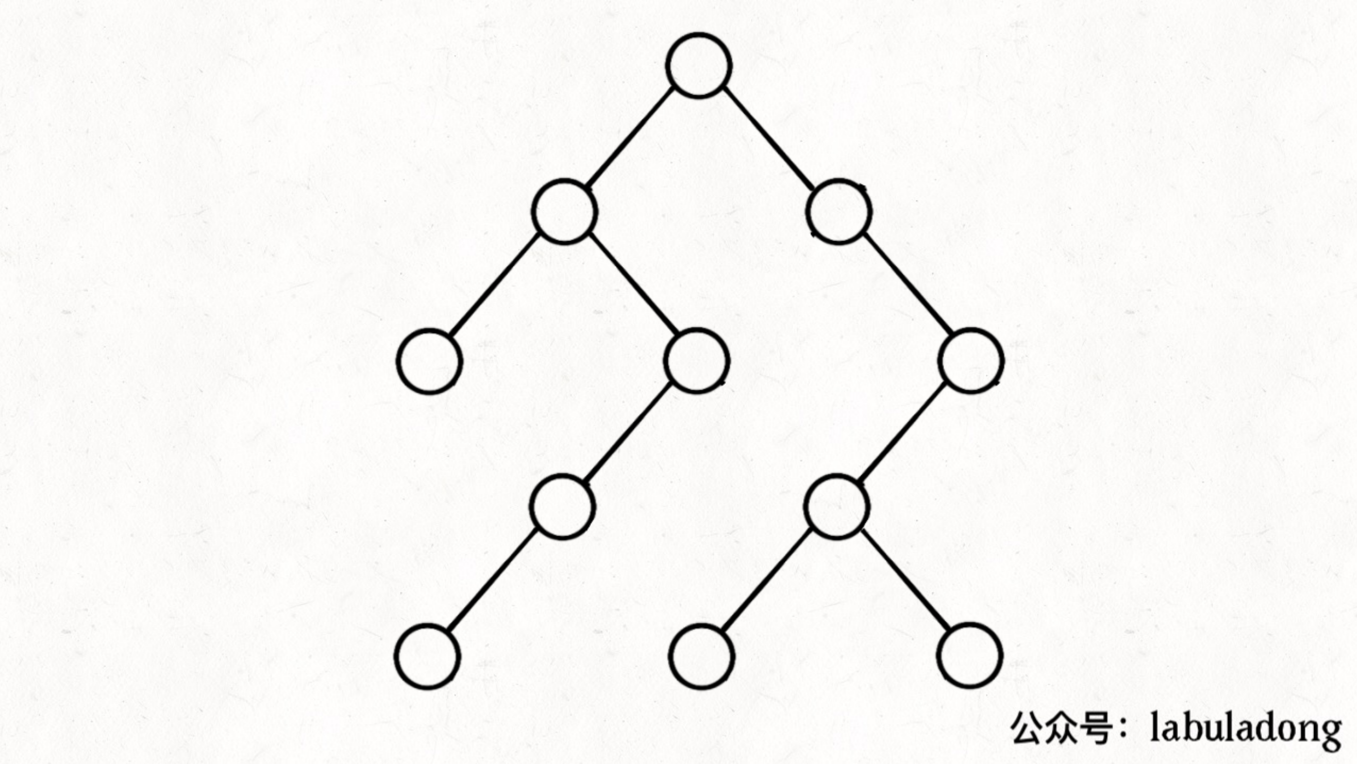
```java
boolean[] onPath;
boolean[] visited;

boolean hasCycle = false;

void traverse(List<Integer>[] graph, int s) {
    if (onPath[s]) {
        // 发现环！！！
        hasCycle = true;
    }
    if (visited[s] || hasCycle) {
        return;
    }
    // 将节点 s 标记为已遍历
    visited[s] = true;
    // 开始遍历节点 s
    onPath[s] = true;
    for (int t : graph[s]) {
        traverse(graph, t);
    }
    // 节点 s 遍历完成
    onPath[s] = false;
}
```

这里就有点回溯算法的味道了，在进入节点 `s` 的时候将 `onPath[s]` 标记为 true，离开时标记回 false，如果发现 `onPath[s]` 已经被标记，说明出现了环。

注意 `visited` 数组和 `onPath` 数组的区别，因为二叉树算是特殊的图，所以用遍历二叉树的过程来理解下这两个数组的区别：



上述 GIF 描述了递归遍历二叉树的过程，在 `visited` 中被标记为 true 的节点用灰色表示，在 `onPath` 中被标记为 true 的节点用绿色表示。

> PS：类比贪吃蛇游戏，`visited` 记录蛇经过过的格子，而 `onPath` 仅仅记录蛇身。`onPath` 用于判断是否成环，类比当贪吃蛇自己咬到自己（成环）的场景。

这样，就可以在遍历图的过程中顺便判断是否存在环了，完整代码如下：

```java
// 记录一次递归堆栈中的节点
boolean[] onPath;
// 记录遍历过的节点，防止走回头路
boolean[] visited;
// 记录图中是否有环
boolean hasCycle = false;

boolean canFinish(int numCourses, int[][] prerequisites) {
    List<Integer>[] graph = buildGraph(numCourses, prerequisites);

    visited = new boolean[numCourses];
    onPath = new boolean[numCourses];

    for (int i = 0; i < numCourses; i++) {
        // 遍历图中的所有节点
        traverse(graph, i);
    }
    // 只要没有循环依赖可以完成所有课程
    return !hasCycle;
}

void traverse(List<Integer>[] graph, int s) {
    if (onPath[s]) {
```

```java
            // 出现环
            hasCycle = true;
        }

        if (visited[s] || hasCycle) {
            // 如果已经找到了环，也不用再遍历了
            return;
        }
        // 前序代码位置
        visited[s] = true;
        onPath[s] = true;
        for (int t : graph[s]) {
            traverse(graph, t);
        }
        // 后序代码位置
        onPath[s] = false;
    }

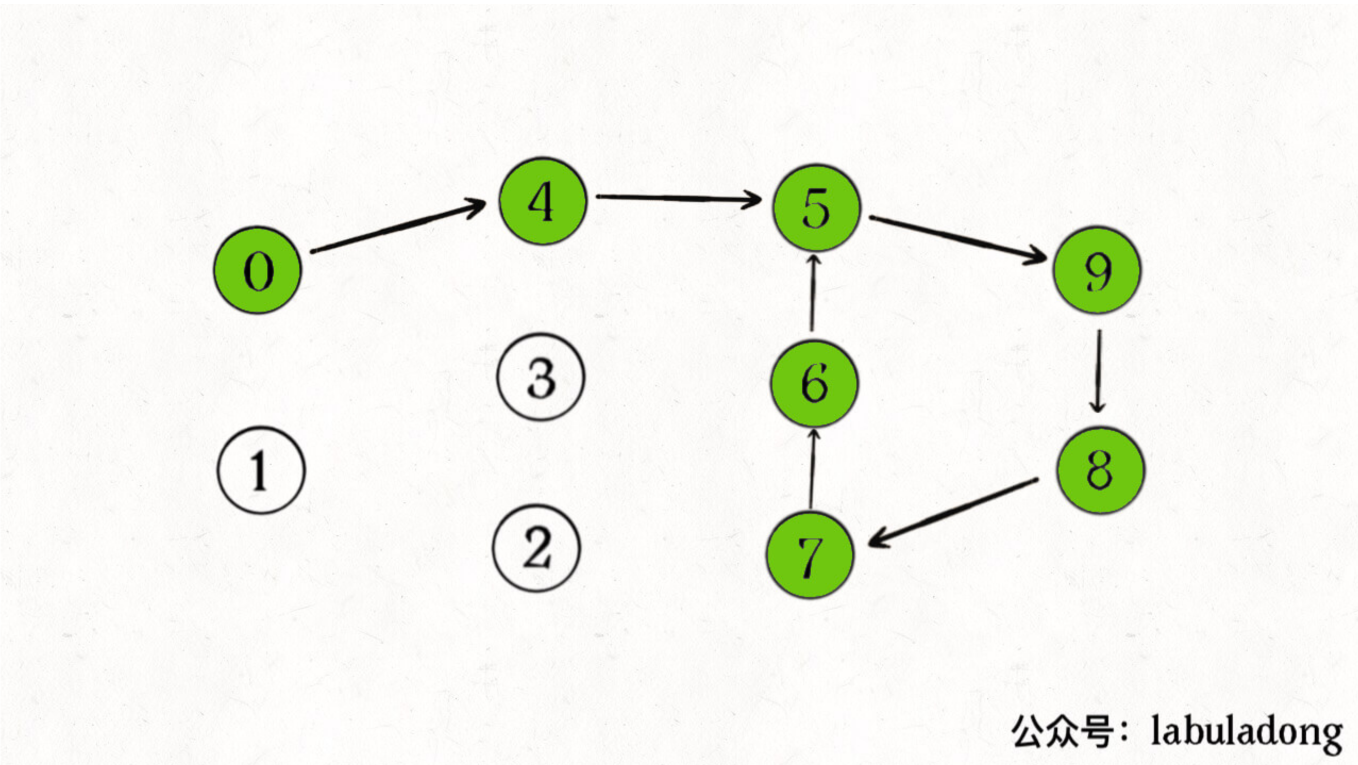
    List<Integer>[] buildGraph(int numCourses, int[][] prerequisites) {
        // 代码见前文
    }
```

这道题就解决了，核心就是判断一幅有向图中是否存在环。

不过如果出题人继续恶心你，让你不仅要判断是否存在环，还要返回这个环具体有哪些节点，怎么办？

你可能说，`onPath` 里面为 true 的索引，不就是组成环的节点编号吗？

不是的，假设下图中绿色的节点是递归的路径，它们在 `onPath` 中的值都是 true，但显然成环的节点只是其中的一部分：



这个问题留给大家思考，我会在公众号留言区置顶正确的答案。

那么接下来，我们来再讲一个经典的图算法：拓扑排序。

拓扑排序算法（DFS 版本）

看下力扣第 210 题「课程表 II」：

210. 课程表 II labuladong 题解 思路

难度 中等 447

现在你总共有 *n* 门课需要选，记为 `0` 到 `n-1`。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们：`[0,1]`

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

示例 1:

```
输入: 2, [[1,0]]
输出: [0,1]
解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1] 。
```

示例 2:

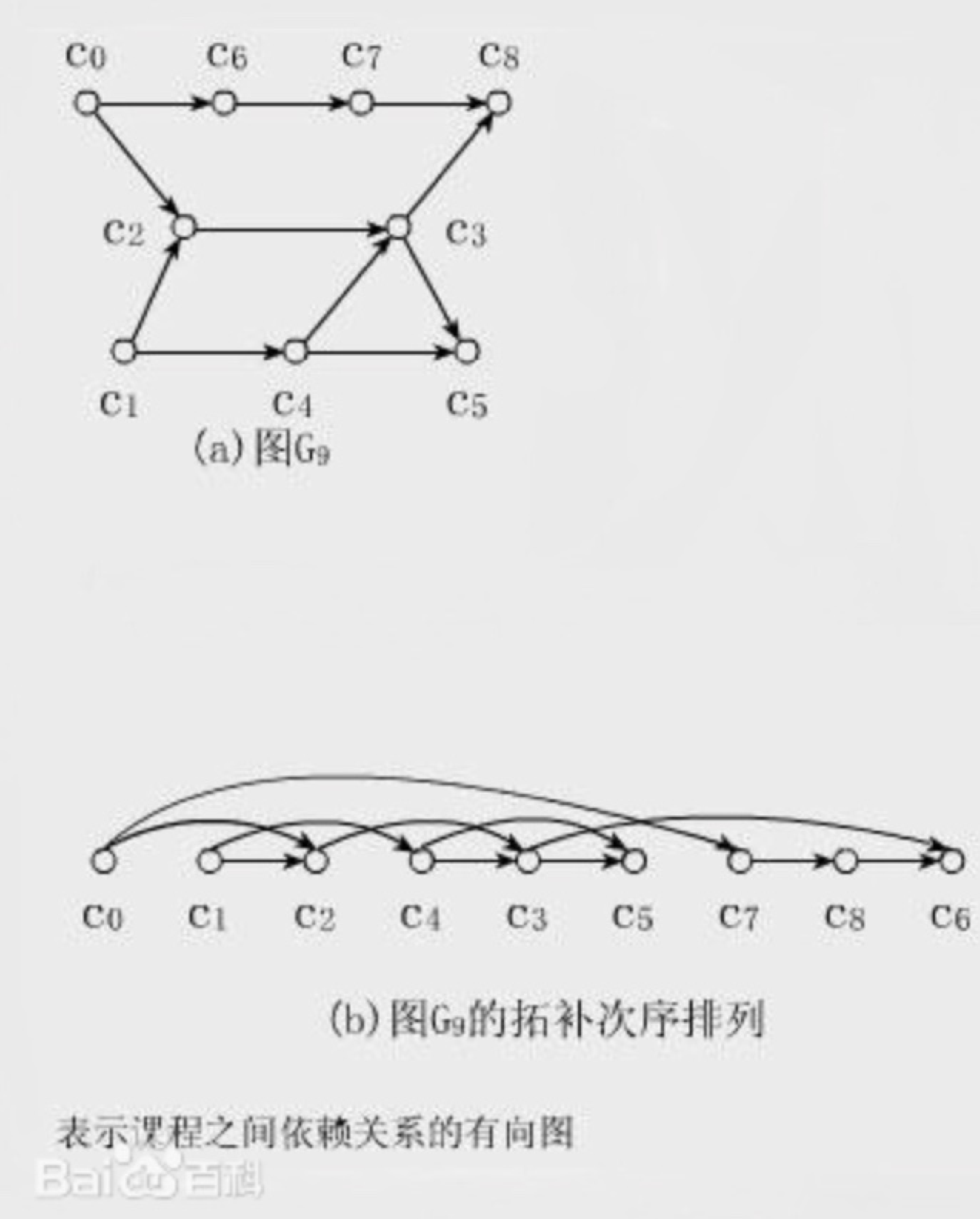
```
输入: 4, [[1,0],[2,0],[3,1],[3,2]]
输出: [0,1,2,3] or [0,2,1,3]
解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。
     因此，一个正确的课程顺序是 [0,1,2,3] 。另一个正确的排序是 [0,2,1,3] 。
```

这道题就是上道题的进阶版，不是仅仅让你判断是否可以完成所有课程，而是进一步让你返回一个合理的上课顺序，保证开始修每个课程时，前置的课程都已经修完。

函数签名如下：

```java
int[] findOrder(int numCourses, int[][] prerequisites);
```

这里我先说一下拓扑排序（Topological Sorting）这个名词，网上搜出来的定义很数学，这里干脆用百度百科的一幅图来让你直观地感受下：



直观地说就是，让你把一幅图「拉平」，而且这个「拉平」的图里面，所有箭头方向都是一致的，比如上图所有箭头都是朝右的。

很显然，如果一幅有向图中存在环，是无法进行拓扑排序的，因为肯定做不到所有箭头方向一致；反过来，如果一幅图是「有向无环图」，那么一定可以进行拓扑排序。

但是我们这道题和拓扑排序有什么关系呢？

其实也不难看出来，如果把课程抽象成节点，课程之间的依赖关系抽象成有向边，那么这幅图的拓扑排序结果就是上课顺序。

首先，我们先判断一下题目输入的课程依赖是否成环，成环的话是无法进行拓扑排序的，所以我们可以复用上一道题的主函数：

```
public int[] findOrder(int numCourses, int[][] prerequisites) {
    if (!canFinish(numCourses, prerequisites)) {
        // 不可能完成所有课程
        return new int[]{};
    }
    // ...
}
```

那么关键问题来了，如何进行拓扑排序？是不是又要秀什么高大上的技巧了？

其实特别简单，将后序遍历的结果进行反转，就是拓扑排序的结果。

> PS：有的读者提到，他在网上看到的拓扑排序算法不用对后序遍历结果进行反转，这是为什么呢？

你确实可以看到这样的解法，原因是他建图的时候对边的定义和我不同。我建的图中箭头方向是「被依赖」关系，比如节点 `1` 指向 `2`，含义是节点 `1` 被节点 `2` 依赖，即做完 `1` 才能去做 `2`，

如果你反过来，把有向边定义为「依赖」关系，那么整幅图中边全部反转，就可以不对后序遍历结果反转。具体来说，就是把我的解法代码中 `graph[from].add(to);` 改成 `graph[to].add(from);` 就可以不反转了。

不过呢，现实中一般都是从初级任务指向进阶任务，所以像我这样把边定义为「被依赖」关系可能比较符合我们的认知习惯。

直接看解法代码吧，在上一题环检测的代码基础上添加了记录后序遍历结果的逻辑：

```
// 记录后序遍历结果
List<Integer> postorder = new ArrayList<>();
// 记录是否存在环
boolean hasCycle = false;
boolean[] visited, onPath;

// 主函数
public int[] findOrder(int numCourses, int[][] prerequisites) {
    List<Integer>[] graph = buildGraph(numCourses, prerequisites);
    visited = new boolean[numCourses];
    onPath = new boolean[numCourses];
    // 遍历图
    for (int i = 0; i < numCourses; i++) {
        traverse(graph, i);
    }
    // 有环图无法进行拓扑排序
```

```java
        if (hasCycle) {
            return new int[]{};
        }
        // 逆后序遍历结果即为拓扑排序结果
        Collections.reverse(postorder);
        int[] res = new int[numCourses];
        for (int i = 0; i < numCourses; i++) {
            res[i] = postorder.get(i);
        }
        return res;
    }

    // 图遍历函数
    void traverse(List<Integer>[] graph, int s) {
        if (onPath[s]) {
            // 发现环
            hasCycle = true;
        }
        if (visited[s] || hasCycle) {
            return;
        }
        // 前序遍历位置
        onPath[s] = true;
        visited[s] = true;
        for (int t : graph[s]) {
            traverse(graph, t);
        }
        // 后序遍历位置
        postorder.add(s);
        onPath[s] = false;
    }

    // 建图函数
    List<Integer>[] buildGraph(int numCourses, int[][] prerequisites) {
        // 代码见前文
    }
```

代码虽然看起来多，但是逻辑应该是很清楚的，只要图中无环，那么我们就调用 `traverse` 函数对图进行 DFS 遍历，记录后序遍历结果，最后把后序遍历结果反转，作为最终的答案。

那么为什么后序遍历的反转结果就是拓扑排序呢？

我这里也避免数学证明，用一个直观地例子来解释，我们就说二叉树，这是我们说过很多次的二叉树遍历框架：

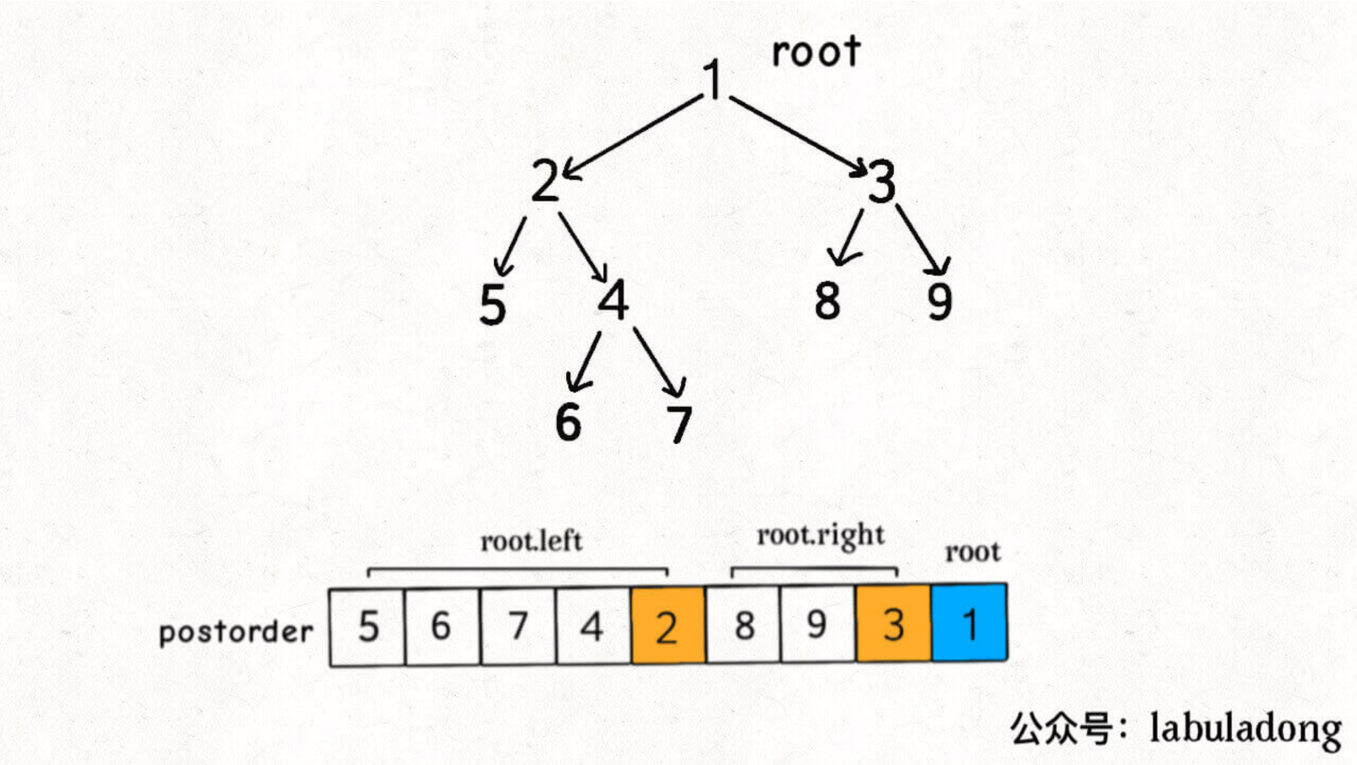
```java
    void traverse(TreeNode root) {
        // 前序遍历代码位置
        traverse(root.left)
        // 中序遍历代码位置
        traverse(root.right)
```

```
        // 后序遍历代码位置
    }
```

二叉树的后序遍历是什么时候？遍历完左右子树之后才会执行后序遍历位置的代码。换句话说，当左右子树的节点都被装到结果列表里面了，根节点才会被装进去。

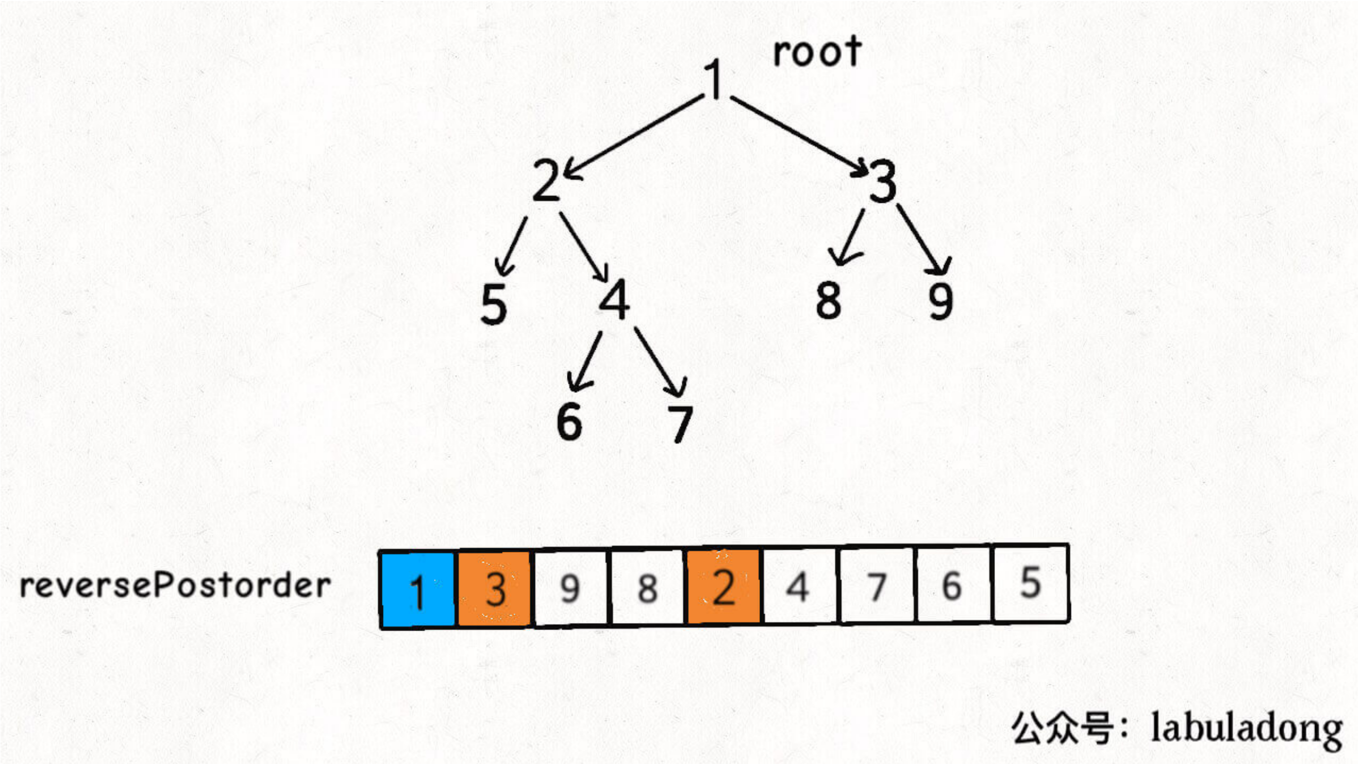
后序遍历的这一特点很重要，之所以拓扑排序的基础是后序遍历，是因为一个任务必须等到它依赖的所有任务都完成之后才能开始开始执行。

你把二叉树理解成一幅有向图，边的方向是由父节点指向子节点，那么就是下图这样：



按照我们的定义，边的含义是「被依赖」关系，那么上图的拓扑排序应该首先是节点 `1`，然后是 `2`，`3`，以此类推。

但显然标准的后序遍历结果不满足拓扑排序，而如果把后序遍历结果反转，就是拓扑排序结果了：



以上，我直观解释了一下为什么「拓扑排序的结果就是反转之后的后序遍历结果」，当然，我的解释并没有严格的数学证明，有兴趣的读者可以自己查一下。

环检测算法（BFS 版本）

刚才讲了用 DFS 算法利用 `onPath` 数组判断是否存在环；也讲了用 DFS 算法利用逆后序遍历进行拓扑排序。

其实 BFS 算法借助 `indegree` 数组记录每个节点的「入度」，也可以实现这两个算法。不熟悉 BFS 算法的读者可阅读前文 BFS 算法核心框架。

所谓「出度」和「入度」是「有向图」中的概念，很直观：如果一个节点 `x` 有 `a` 条边指向别的节点，同时被 `b` 条边所指，则称节点 `x` 的出度为 `a`，入度为 `b`。

先说环检测算法，直接看 BFS 的解法代码：

```
// 主函数
public boolean canFinish(int numCourses, int[][] prerequisites) {
    // 建图，有向边代表「被依赖」关系
    List<Integer>[] graph = buildGraph(numCourses, prerequisites);
    // 构建入度数组
    int[] indgree = new int[numCourses];
    for (int[] edge : prerequisites) {
        int from = edge[1], to = edge[0];
        // 节点 to 的入度加一
        indgree[to]++;
    }

    // 根据入度初始化队列中的节点
    Queue<Integer> q = new LinkedList<>();
    for (int i = 0; i < numCourses; i++) {
```

```java
            if (indgree[i] == 0) {
                // 节点 i 没有入度，即没有依赖的节点
                // 可以作为拓扑排序的起点，加入队列
                q.offer(i);
            }
        }

        // 记录遍历的节点个数
        int count = 0;
        // 开始执行 BFS 循环
        while (!q.isEmpty()) {
            // 弹出节点 cur，并将它指向的节点的入度减一
            int cur = q.poll();
            count++;
            for (int next : graph[cur]) {
                indgree[next]--;
                if (indgree[next] == 0) {
                    // 如果入度变为 0，说明 next 依赖的节点都已被遍历
                    q.offer(next);
                }
            }
        }

        // 如果所有节点都被遍历过，说明不成环
        return count == numCourses;
    }


    // 建图函数
    List<Integer>[] buildGraph(int n, int[][] edges) {
        // 见前文
    }
```

我先总结下这段 BFS 算法的思路：

1、构建邻接表，和之前一样，边的方向表示「被依赖」关系。

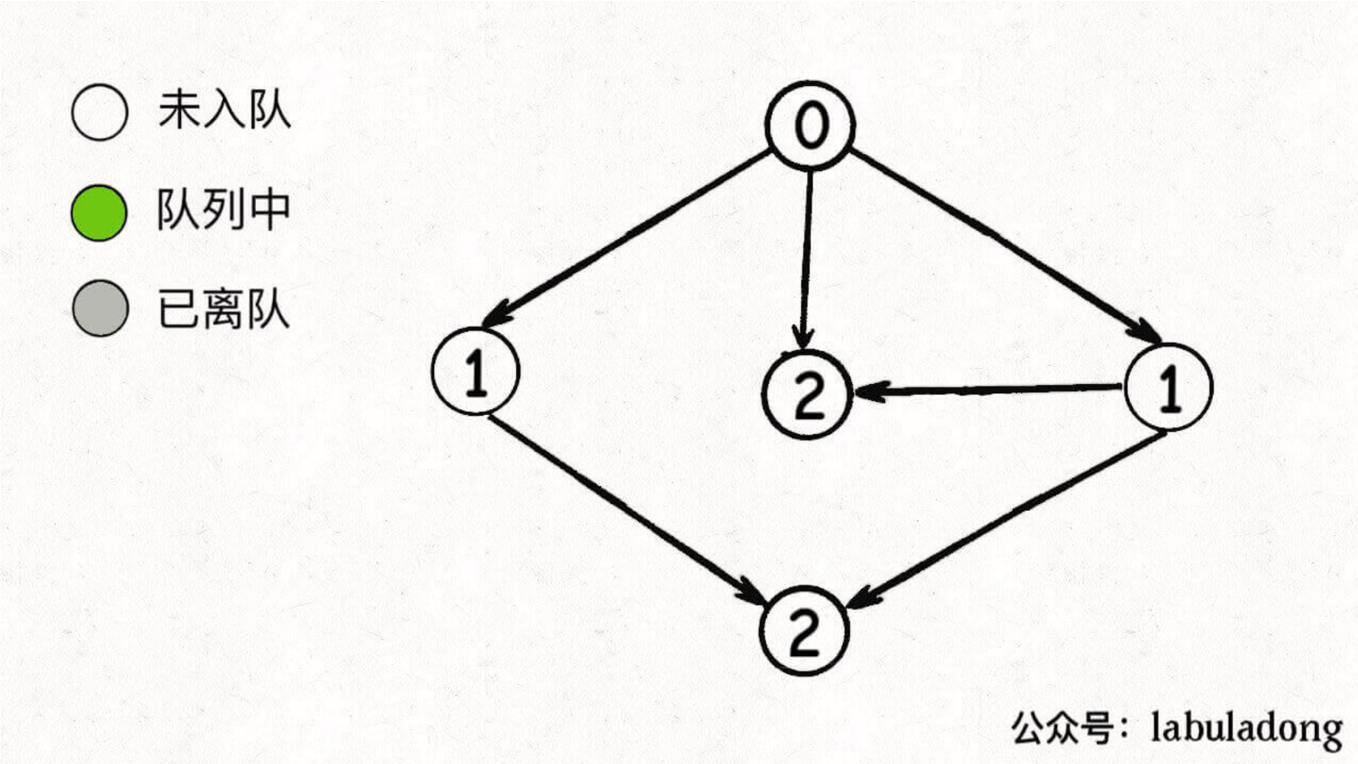
2、构建一个 `indegree` 数组记录每个节点的入度，即 `indegree[i]` 记录节点 `i` 的入度。

3、对 BFS 队列进行初始化，将入度为 0 的节点首先装入队列。

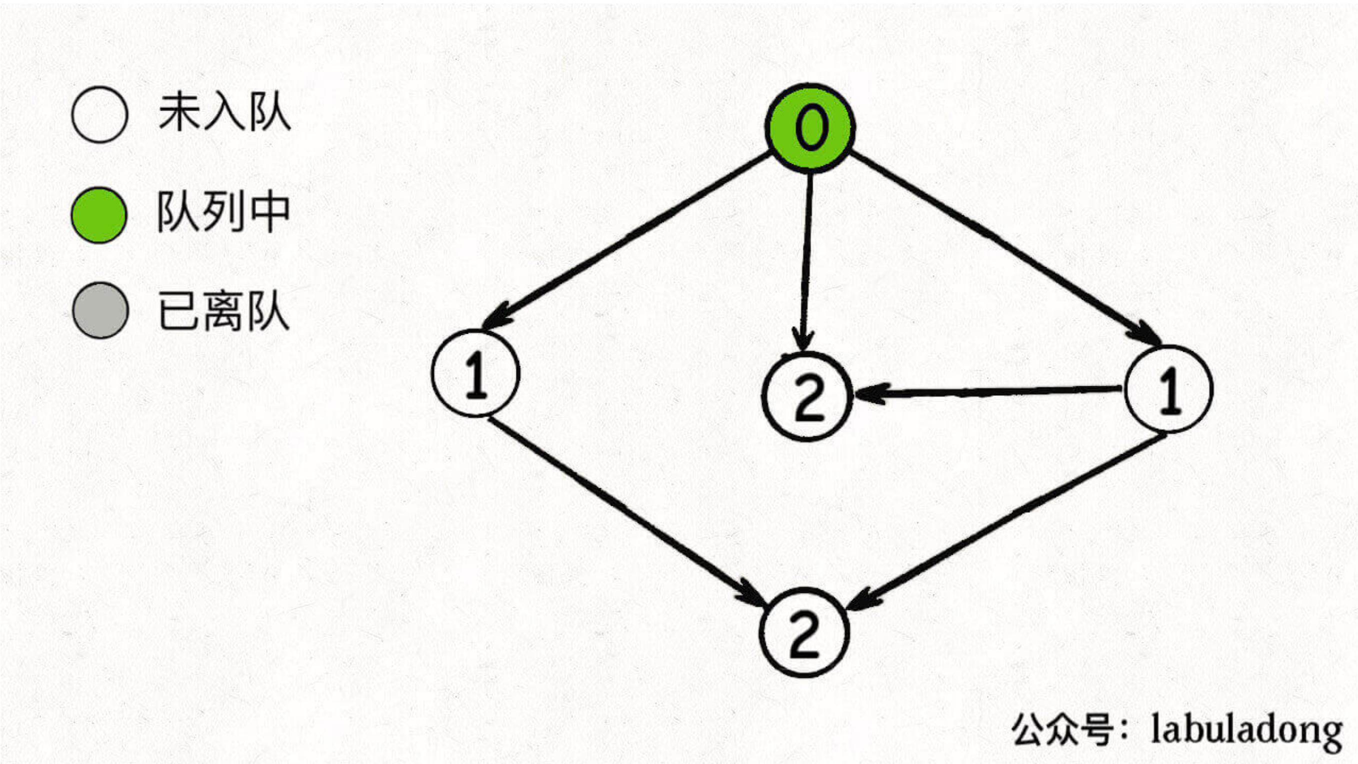
4、开始执行 BFS 循环，不断弹出队列中的节点，减少相邻节点的入度，并将入度变为 0 的节点加入队列。

5、如果最终所有节点都被遍历过（`count` 等于节点数），则说明不存在环，反之则说明存在环。

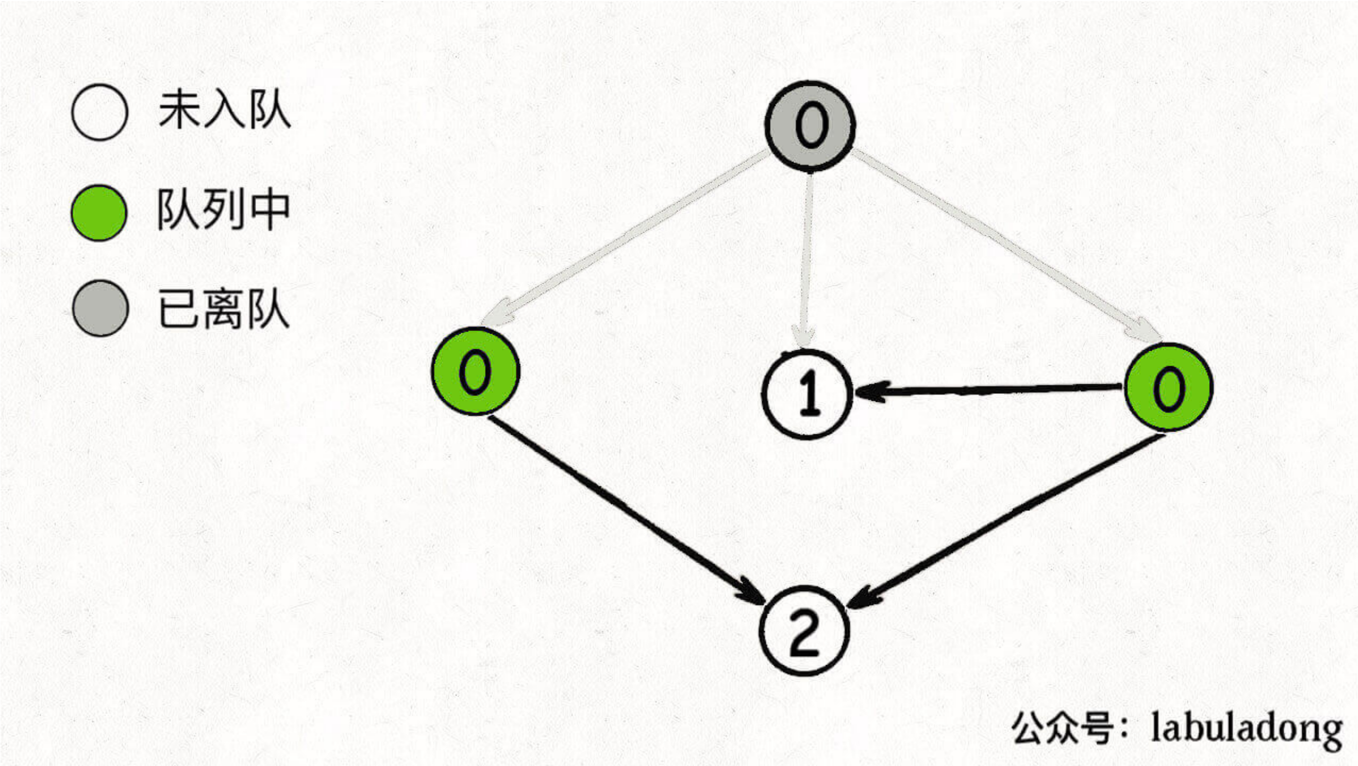
我画个图你就容易理解了，比如下面这幅图，节点中的数字代表该节点的入度：



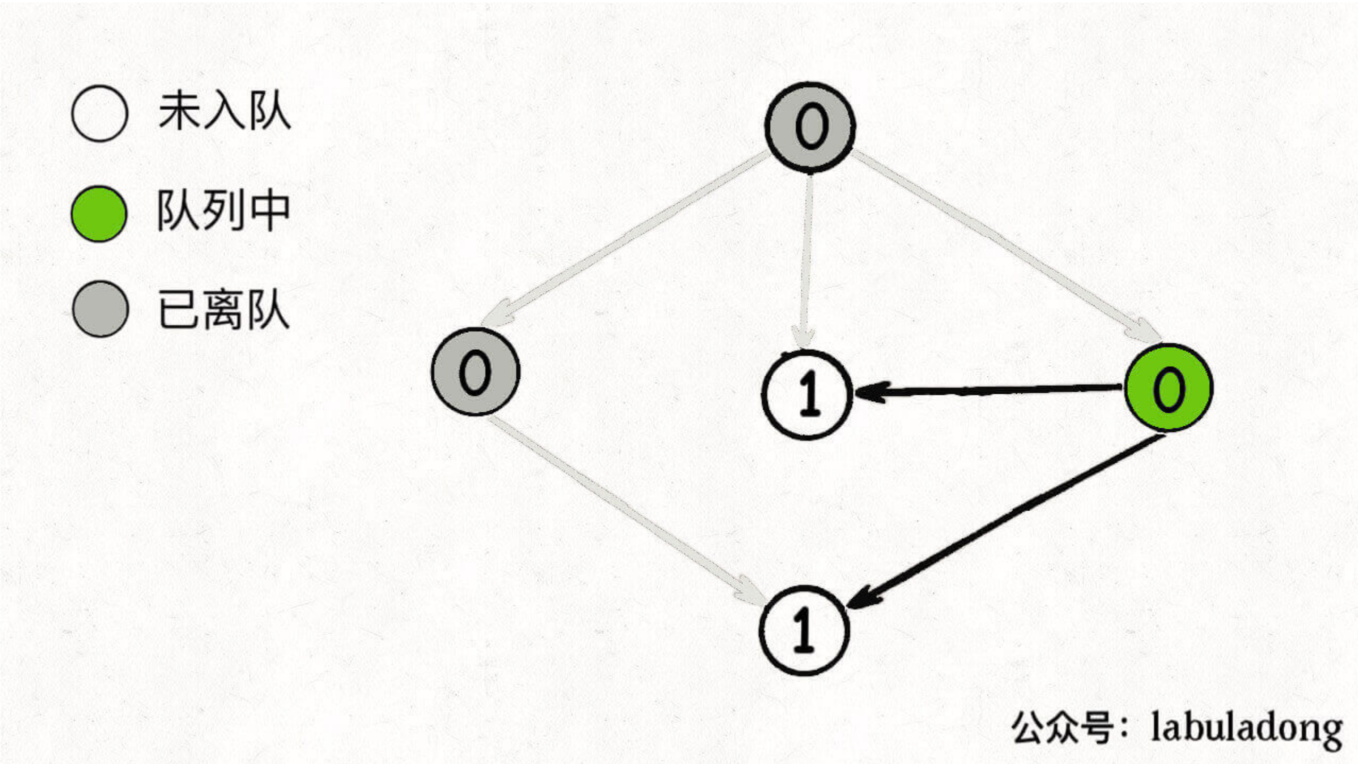
队列进行初始化后，入度为 0 的节点首先被加入队列：



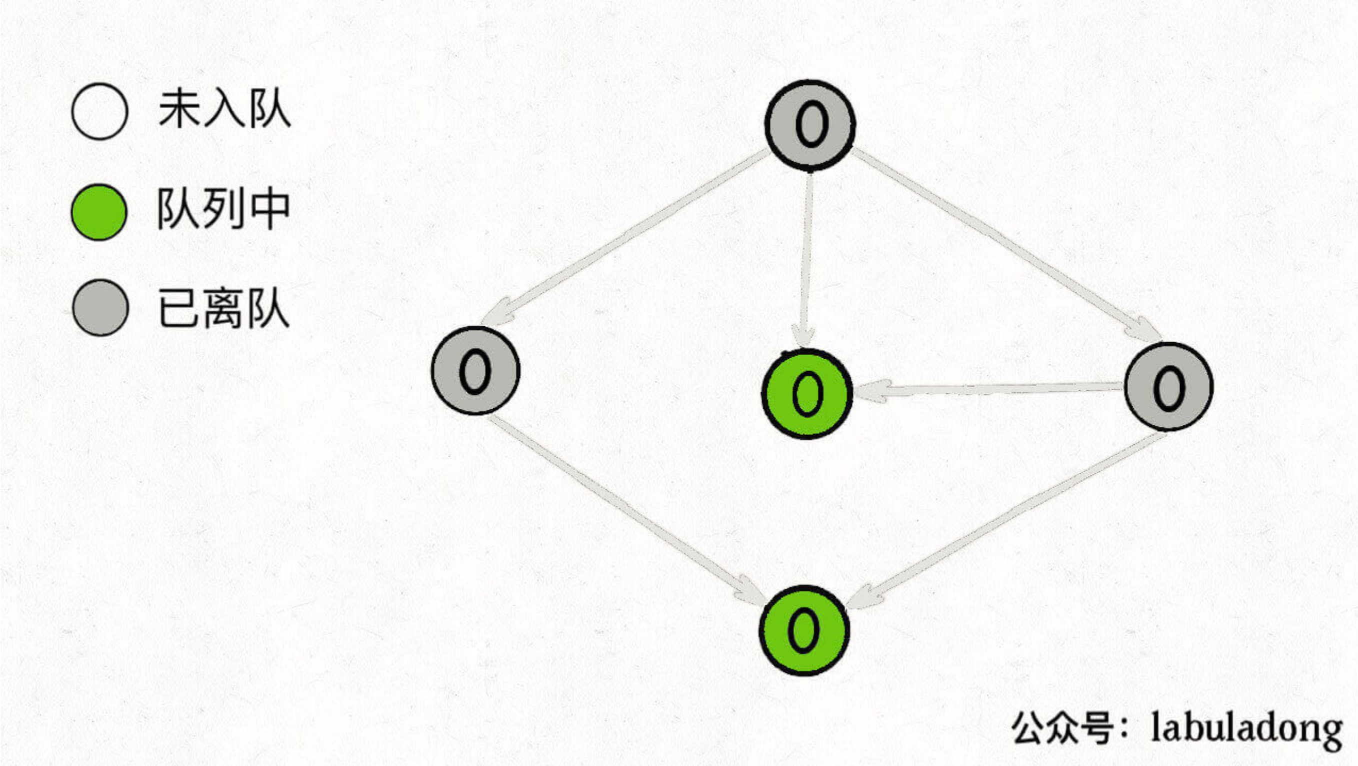
开始执行 BFS 循环，从队列中弹出一个节点，减少相邻节点的入度，同时将新产生的入度为 0 的节点加入队列：



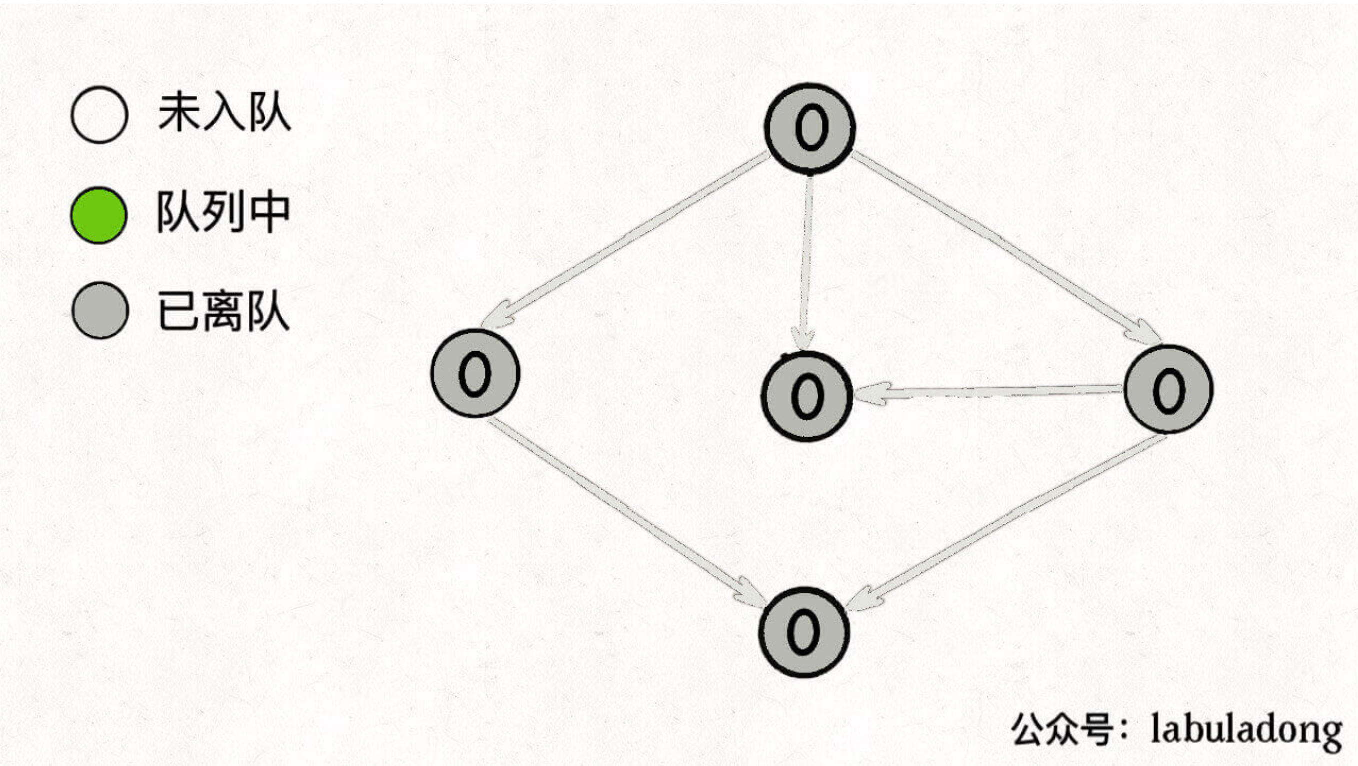
继续从队列弹出节点，并减少相邻节点的入度，这一次没有新产生的入度为 0 的节点：



继续从队列弹出节点，并减少相邻节点的入度，同时将新产生的入度为 0 的节点加入队列：



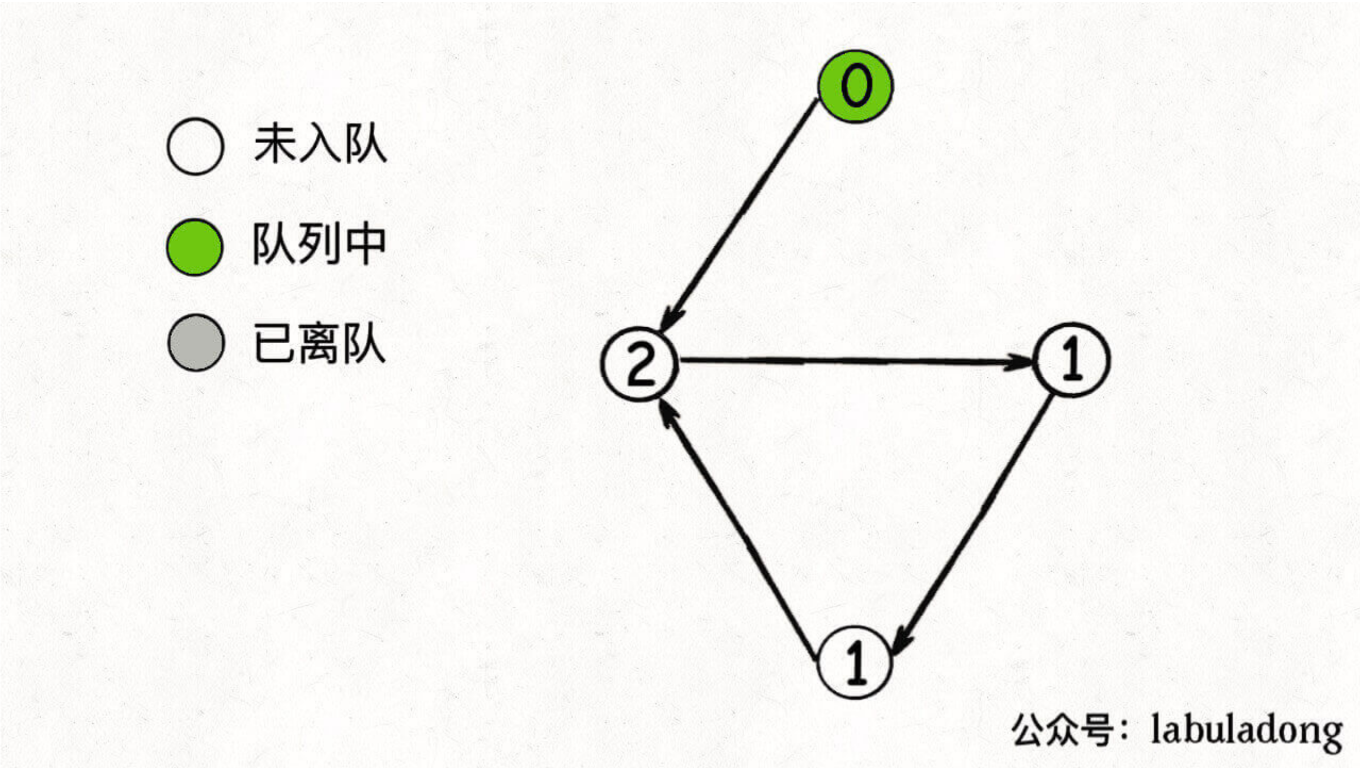
继续弹出节点，直到队列为空：



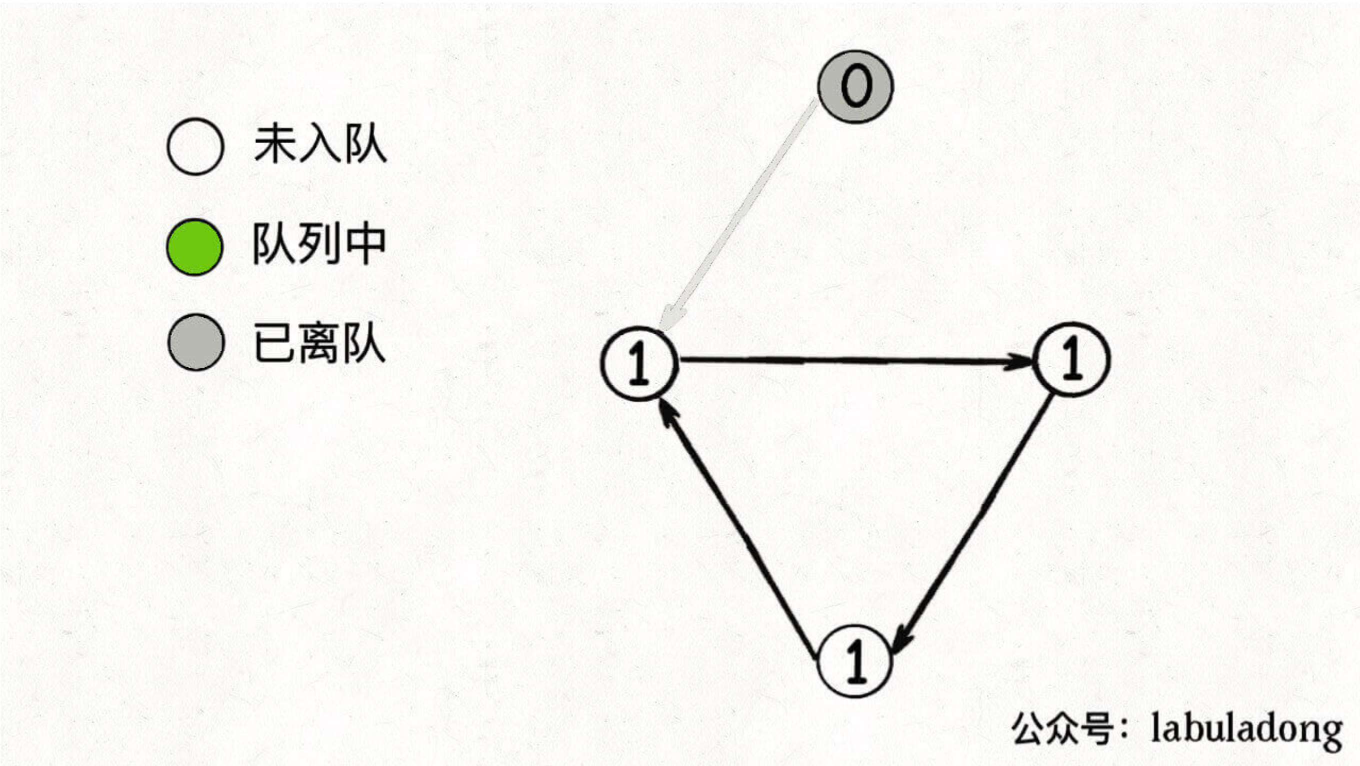
这时候，所有节点都被遍历过一遍，也就说明图中不存在环。

反过来说，如果按照上述逻辑执行 BFS 算法，存在节点没有被遍历，则说明成环。

比如下面这种情况，队列中最初只有一个入度为 0 的节点：



当弹出这个节点并减小相邻节点的入度之后队列为空，但并没有产生新的入度为 0 的节点加入队列，所以 BFS 算法终止：

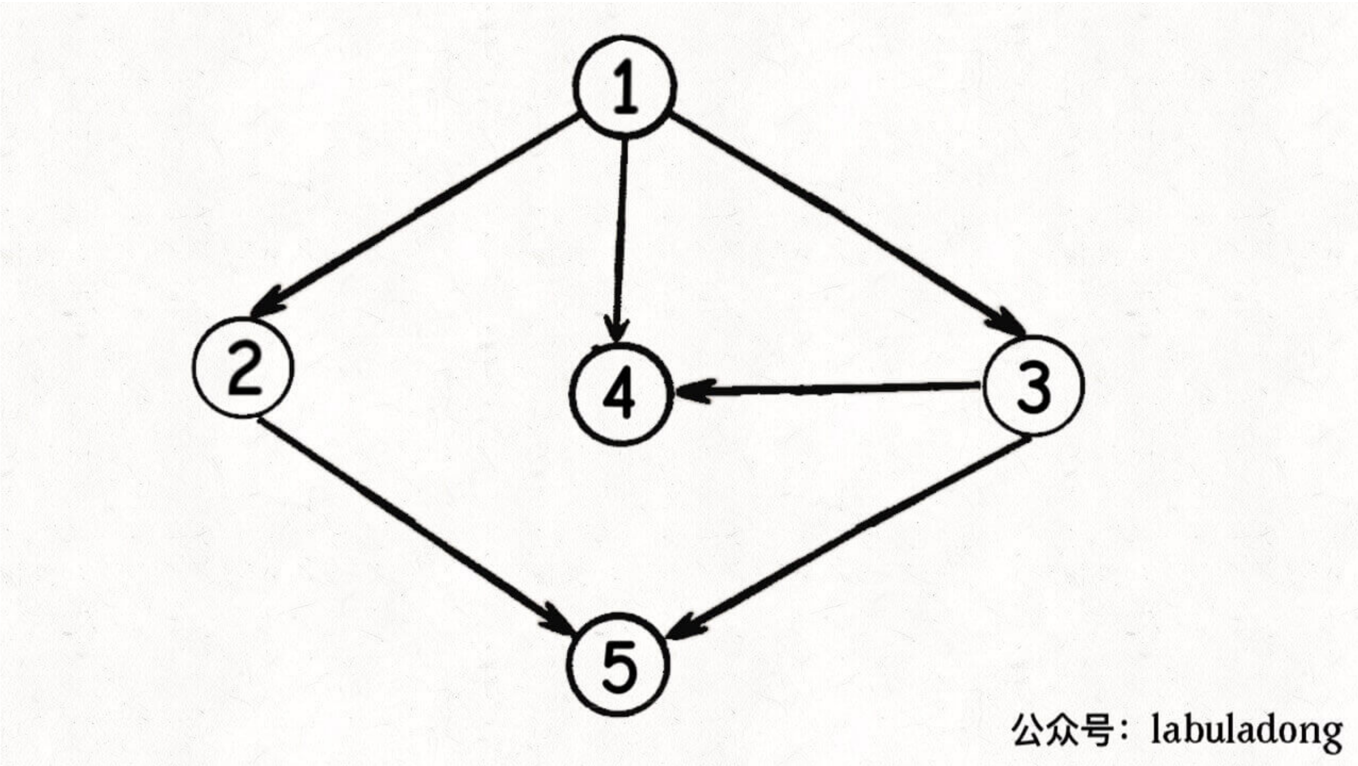


你看到了，如果存在节点没有被遍历，那么说明图中存在环，现在回头去看 BFS 的代码，你应该就很容易理解其中的逻辑了。

拓扑排序算法（BFS 版本）

如果你能看懂 BFS 版本的环检测算法，那么就很容易得到 BFS 版本的拓扑排序算法，因为节点的遍历顺序就是拓扑排序的结果。

比如刚才举的第一个例子，下图每个节点中的值即入队的顺序：



显然，这个顺序就是一个可行的拓扑排序结果。

所以，我们稍微修改一下 BFS 版本的环检测算法，记录节点的遍历顺序即可得到拓扑排序的结果：

```java
// 主函数
public int[] findOrder(int numCourses, int[][] prerequisites) {
    // 建图，和环检测算法相同
    List<Integer>[] graph = buildGraph(numCourses, prerequisites);
    // 计算入度，和环检测算法相同
    int[] indgree = new int[numCourses];
    for (int[] edge : prerequisites) {
        int from = edge[1], to = edge[0];
        indgree[to]++;
    }

    // 根据入度初始化队列中的节点，和环检测算法相同
    Queue<Integer> q = new LinkedList<>();
    for (int i = 0; i < numCourses; i++) {
        if (indgree[i] == 0) {
            q.offer(i);
        }
    }

    // 记录拓扑排序结果
    int[] res = new int[numCourses];
    // 记录遍历节点的顺序（索引）
    int count = 0;
    // 开始执行 BFS 算法
    while (!q.isEmpty()) {
        int cur = q.poll();
```

```java
            // 弹出节点的顺序即为拓扑排序结果
            res[count] = cur;
            count++;
            for (int next : graph[cur]) {
                indgree[next]--;
                if (indgree[next] == 0) {
                    q.offer(next);
                }
            }
        }

        if (count != numCourses) {
            // 存在环，拓扑排序不存在
            return new int[]{};
        }

        return res;
    }

    // 建图函数
    List<Integer>[] buildGraph(int n, int[][] edges) {
        // 见前文
    }
```

按道理，图的遍历 都需要 `visited` 数组防止走回头路，这里的 BFS 算法其实是通过 `indegree` 数组实现的 `visited` 数组的作用，只有入度为 0 的节点才能入队，从而保证不会出现死循环。

好了，到这里环检测算法、拓扑排序算法的 BFS 实现也讲完了，继续留一个思考题：

对于 BFS 的环检测算法，如果问你形成环的节点具体是哪些，你应该如何实现呢？

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

二分图判定

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong



学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

785. 判断二分图（中等）

886. 可能的二分法（中等）

我之前写了好几篇图论相关的文章：

图遍历算法

名流问题

并查集算法计算连通分量

环检测和拓扑排序

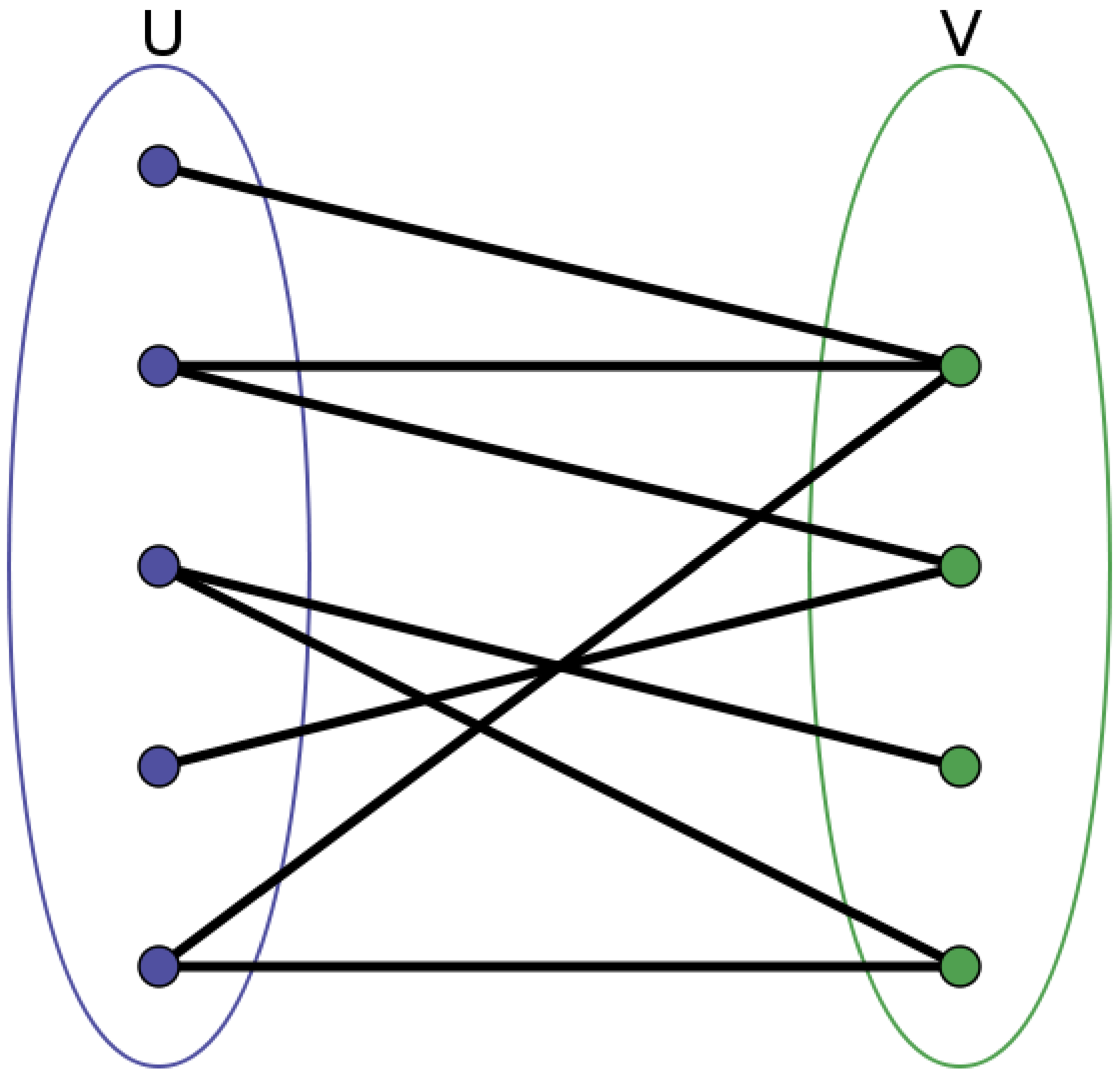
Dijkstra 最短路径算法

今天继续来讲一个经典图论算法：二分图判定。

二分图简介

在讲二分图的判定算法之前，我们先来看下百度百科对「二分图」的定义：

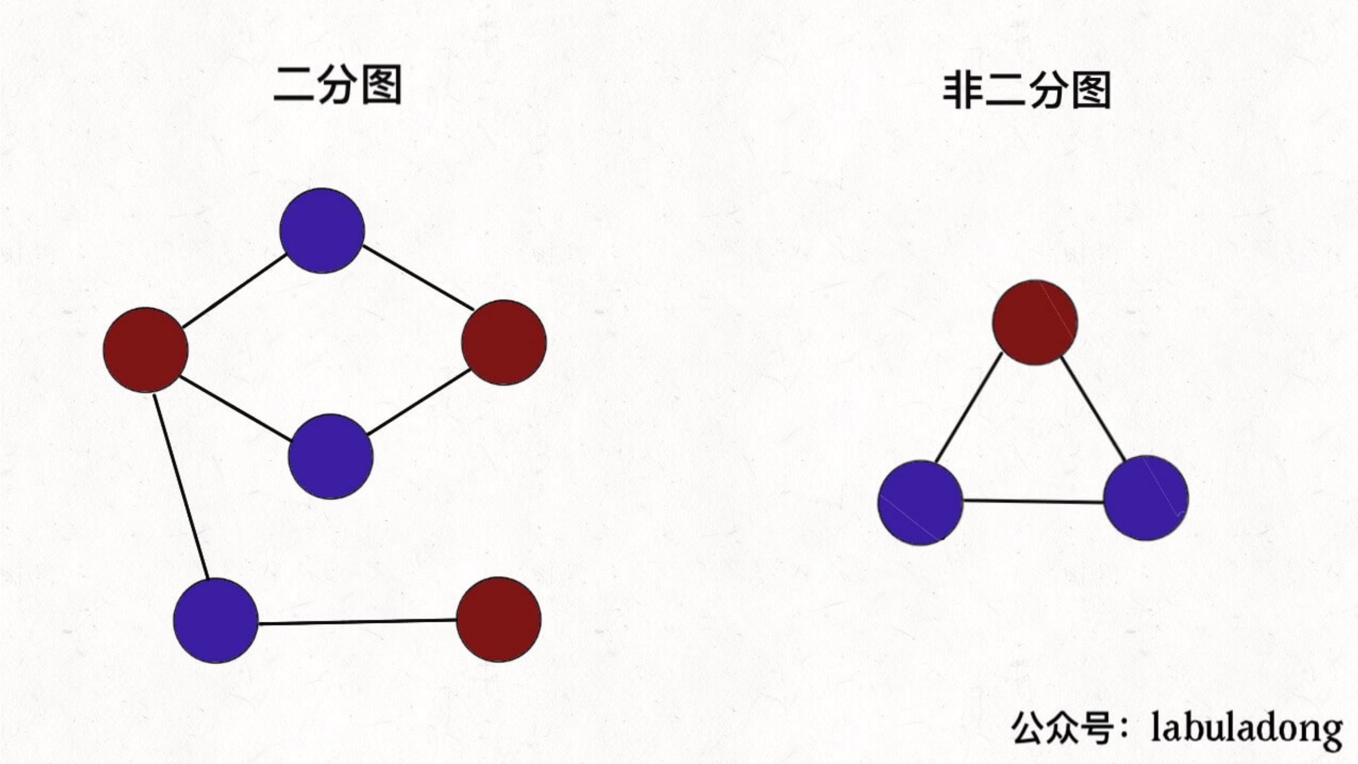
> 二分图的顶点集可分割为两个互不相交的子集，图中每条边依附的两个顶点都分属于这两个子集，且两个子集内的顶点不相邻。



其实图论里面很多术语的定义都比较拗口，不容易理解。我们甭看这个死板的定义了，来玩个游戏吧：

给你一幅「图」，请你用两种颜色将图中的所有顶点着色，且使得任意一条边的两个端点的颜色都不相同，你能做到吗？

这就是图的「双色问题」，其实这个问题就等同于二分图的判定问题，如果你能够成功地将图染色，那么这幅图就是一幅二分图，反之则不是：



在具体讲解二分图判定算法之前，我们先来说说计算机大佬们闲着无聊解决双色问题的目的是什么。

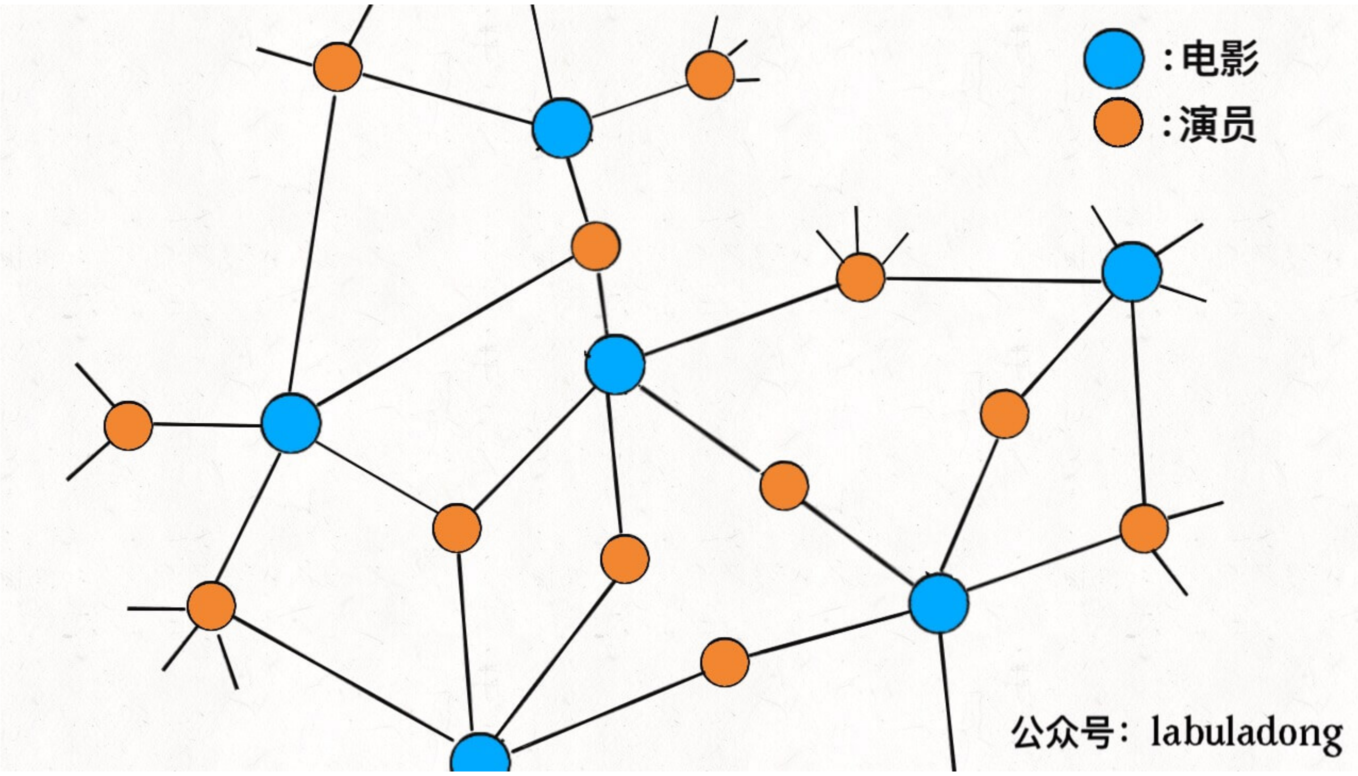
首先，二分图作为一种特殊的图模型，会被很多高级图算法（比如最大流算法）用到，不过这些高级算法我们不是特别有必要去掌握，有兴趣的读者可以自行搜索。

从简单实用的角度来看，二分图结构在某些场景可以更高效地存储数据。

比如前文 介绍《算法 4》 文章中的例子，如何存储电影演员和电影之间的关系?

如果用哈希表存储，需要两个哈希表分别存储「每个演员到电影列表」的映射和「每部电影到演员列表」的映射。

但如果用「图」结构存储，将电影和参演的演员连接，很自然地就成为了一幅二分图：



每个电影节点的相邻节点就是参演该电影的所有演员，每个演员的相邻节点就是该演员参演过的所有电影，非常方便直观。

类比这个例子，其实生活中不少实体的关系都能自然地形成二分图结构，所以在某些场景下图结构也可以作为存储键值对的数据结构（符号表）。

好了，接下来进入正题，说说如何判定一幅图是否是二分图。

二分图判定思路

判定二分图的算法很简单，就是用代码解决「双色问题」。

说白了就是遍历一遍图，一边遍历一边染色，看看能不能用两种颜色给所有节点染色，且相邻节点的颜色都不相同。

既然说到遍历图，也不涉及最短路径之类的，当然是 DFS 算法和 BFS 皆可了，DFS 算法相对更常用些，所以我们先来看看如何用 DFS 算法判定双色图。

首先，基于 学习数据结构和算法的框架思维 写出图的遍历框架：

```
/* 二叉树遍历框架 */
void traverse(TreeNode root) {
    if (root == null) return;
    traverse(root.left);
    traverse(root.right);
}

/* 多叉树遍历框架 */
void traverse(Node root) {
    if (root == null) return;
    for (Node child : root.children)
```

```
        traverse(child);
}

/* 图遍历框架 */
boolean[] visited;
void traverse(Graph graph, int v) {
    // 防止走回头路进入死循环
    if (visited[v]) return;
    // 前序遍历位置，标记节点 v 已访问
    visited[v] = true;
    for (TreeNode neighbor : graph.neighbors(v))
        traverse(graph, neighbor);
}
```

因为图中可能存在环，所以用 `visited` 数组防止走回头路。

这里可以看到我习惯把 return 语句都放在函数开头，因为一般 return 语句都是 base case，集中放在一起可以让算法结构更清晰。

其实，如果你愿意，也可以把 if 判断放到其它地方，比如图遍历框架可以稍微改改：

```
/* 图遍历框架 */
boolean[] visited;
void traverse(Graph graph, int v) {
    // 前序遍历位置，标记节点 v 已访问
    visited[v] = true;
    for (int neighbor : graph.neighbors(v)) {
        if (!visited[neighbor]) {
            // 只遍历没标记过的相邻节点
            traverse(graph, neighbor);
        }
    }
}
```

这种写法把对 `visited` 的判断放到递归调用之前，和之前的写法唯一的不同就是，你需要保证调用 `traverse(v)` 的时候，`visited[v] == false`。

为什么要特别说这种写法呢？因为我们判断二分图的算法会用到这种写法。

回顾一下二分图怎么判断，其实就是让 `traverse` 函数一边遍历节点，一边给节点染色，尝试让每对相邻节点的颜色都不一样。

所以，判定二分图的代码逻辑可以这样写：

```
/* 图遍历框架 */
void traverse(Graph graph, boolean[] visited, int v) {
    visited[v] = true;
    // 遍历节点 v 的所有相邻节点 neighbor
    for (int neighbor : graph.neighbors(v)) {
```

```
            if (!visited[neighbor]) {
                // 相邻节点 neighbor 没有被访问过
                // 那么应该给节点 neighbor 涂上和节点 v 不同的颜色
                traverse(graph, visited, neighbor);
            } else {
                // 相邻节点 neighbor 已经被访问过
                // 那么应该比较节点 neighbor 和节点 v 的颜色
                // 若相同，则此图不是二分图
            }
        }
    }
```

如果你能看懂上面这段代码，就能写出二分图判定的具体代码了，接下来看两道具体的算法题来实操一下。

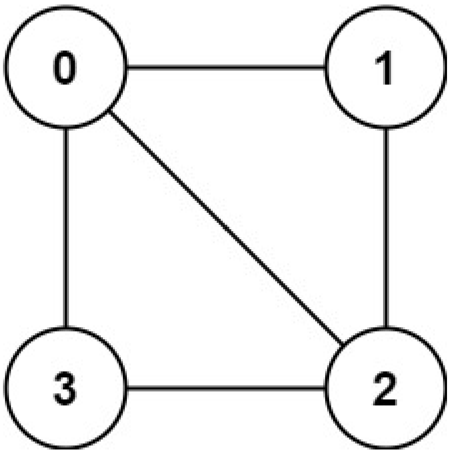
题目实践

力扣第 785 题「判断二分图」就是原题，题目给你输入一个 邻接表 表示一幅无向图，请你判断这幅图是否是二分图。

函数签名如下：

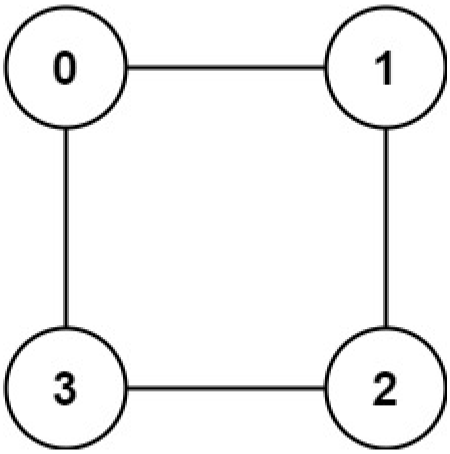
```
boolean isBipartite(int[][] graph);
```

比如题目给的例子，输入的邻接表 `graph = [[1,2,3],[0,2],[0,1,3],[0,2]]`，也就是这样一幅图：



显然无法对节点着色使得每两个相邻节点的颜色都不相同，所以算法返回 false。

但如果输入 `graph = [[1,3],[0,2],[1,3],[0,2]]`，也就是这样一幅图：



如果把节点 `{0, 2}` 涂一个颜色，节点 `{1, 3}` 涂另一个颜色，就可以解决「双色问题」，所以这是一幅二分图，算法返回 true。

结合之前的代码框架，我们可以额外使用一个 `color` 数组来记录每个节点的颜色，从而写出解法代码：

```
// 记录图是否符合二分图性质
private boolean ok = true;
// 记录图中节点的颜色，false 和 true 代表两种不同颜色
private boolean[] color;
// 记录图中节点是否被访问过
private boolean[] visited;

// 主函数，输入邻接表，判断是否是二分图
public boolean isBipartite(int[][] graph) {
    int n = graph.length;
    color =  new boolean[n];
    visited =  new boolean[n];
    // 因为图不一定是联通的，可能存在多个子图
    // 所以要把每个节点都作为起点进行一次遍历
    // 如果发现任何一个子图不是二分图，整幅图都不算二分图
    for (int v = 0; v < n; v++) {
        if (!visited[v]) {
            traverse(graph, v);
        }
    }
    return ok;
}

// DFS 遍历框架
private void traverse(int[][] graph, int v) {
    // 如果已经确定不是二分图了，就不用浪费时间再递归遍历了
    if (!ok) return;

    visited[v] = true;
    for (int w : graph[v]) {
        if (!visited[w]) {
            // 相邻节点 w 没有被访问过
            // 那么应该给节点 w 涂上和节点 v 不同的颜色
            color[w] = !color[v];
            // 继续遍历 w
```

```
                traverse(graph, w);
            } else {
                // 相邻节点 w 已经被访问过
                // 根据 v 和 w 的颜色判断是否是二分图
                if (color[w] == color[v]) {
                    // 若相同，则此图不是二分图
                    ok = false;
                }
            }
        }
    }
```

这就是解决「双色问题」的代码，如果能成功对整幅图染色，则说明这是一幅二分图，否则就不是二分图。

接下来看一下 BFS 算法的逻辑：

```
// 记录图是否符合二分图性质
private boolean ok = true;
// 记录图中节点的颜色，false 和 true 代表两种不同颜色
private boolean[] color;
// 记录图中节点是否被访问过
private boolean[] visited;

public boolean isBipartite(int[][] graph) {
    int n = graph.length;
    color =  new boolean[n];
    visited =  new boolean[n];

    for (int v = 0; v < n; v++) {
        if (!visited[v]) {
            // 改为使用 BFS 函数
            bfs(graph, v);
        }
    }

    return ok;
}

// 从 start 节点开始进行 BFS 遍历
private void bfs(int[][] graph, int start) {
    Queue<Integer> q = new LinkedList<>();
    visited[start] = true;
    q.offer(start);

    while (!q.isEmpty() && ok) {
        int v = q.poll();
        // 从节点 v 向所有相邻节点扩散
        for (int w : graph[v]) {
            if (!visited[w]) {
                // 相邻节点 w 没有被访问过
                // 那么应该给节点 w 涂上和节点 v 不同的颜色
                color[w] = !color[v];
```

```
                // 标记 w 节点，并放入队列
                visited[w] = true;
                q.offer(w);
            } else {
                // 相邻节点 w 已经被访问过
                // 根据 v 和 w 的颜色判断是否是二分图
                if (color[w] == color[v]) {
                    // 若相同，则此图不是二分图
                    ok = false;
                }
            }
        }
    }
}
```

核心逻辑和刚才实现的 traverse 函数（DFS 算法）完全一样，也是根据相邻节点 v 和 w 的颜色来进行判断的。关于 BFS 算法框架的探讨，详见前文 BFS 算法框架 和 Dijkstra 算法模板，这里就不展开了。

最后再来看看力扣第 886 题「可能的二分法」：

886. 可能的二分法 labuladong 题解 思路

难度 中等 138

给定一组 N 人（编号为 1, 2, ..., N），我们想把每个人分进**任意**大小的两组。

每个人都可能讨厌其他人，那么他们不应该属于同一组。

形式上，如果 dislikes[i] = [a, b]，表示 a 和 b 互相讨厌对方，不应该把他们分进同一组。

当可以将所有人分进两组时，返回 true；否则返回 false。

示例 1：

```
输入：N = 4, dislikes = [[1,2],[1,3],[2,4]]
输出：true
解释：group1 [1,4], group2 [2,3]
```

示例 2：

```
输入：N = 3, dislikes = [[1,2],[1,3],[2,3]]
输出：false
```

函数签名如下：

```
boolean possibleBipartition(int n, int[][] dislikes);
```

其实这题考察的就是二分图的判定：

如果你把每个人看做图中的节点，相互讨厌的关系看做图中的边，那么 `dislikes` 数组就可以构成一幅图；

又因为题目说互相讨厌的人不能放在同一组里，相当于图中的所有相邻节点都要放进两个不同的组；

那就回到了「双色问题」，如果能够用两种颜色着色所有节点，且相邻节点颜色都不同，那么你按照颜色把这些节点分成两组不就行了嘛。

所以解法就出来了，我们把 `dislikes` 构造成一幅图，然后执行二分图的判定算法即可：

```
private boolean ok = true;
private boolean[] color;
private boolean[] visited;

public boolean possibleBipartition(int n, int[][] dislikes) {
    // 图节点编号从 1 开始
    color = new boolean[n + 1];
    visited = new boolean[n + 1];
    // 转化成邻接表表示图结构
    List<Integer>[] graph = buildGraph(n, dislikes);

    for (int v = 1; v <= n; v++) {
        if (!visited[v]) {
            traverse(graph, v);
        }
    }

    return ok;
}

// 建图函数
private List<Integer>[] buildGraph(int n, int[][] dislikes) {
    // 图节点编号为 1...n
    List<Integer>[] graph = new LinkedList[n + 1];
    for (int i = 1; i <= n; i++) {
        graph[i] = new LinkedList<>();
    }
    for (int[] edge : dislikes) {
        int v = edge[1];
        int w = edge[0];
        // 「无向图」相当于「双向图」
        // v -> w
        graph[v].add(w);
        // w -> v
        graph[w].add(v);
    }
```

```java
        return graph;
    }

    // 和之前的 traverse 函数完全相同
    private void traverse(List<Integer>[] graph, int v) {
        if (!ok) return;
        visited[v] = true;
        for (int w : graph[v]) {
            if (!visited[w]) {
                color[w] = !color[v];
                traverse(graph, w);
            } else {
                if (color[w] == color[v]) {
                    ok = false;
                }
            }
        }
    }
```

至此，这道题也使用 DFS 算法解决了，如果你想用 BFS 算法，和之前写的解法是完全一样的，可以自己尝试实现。

二分图的判定算法就讲到这里，更多二分图的高级算法，敬请期待。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



并查集算法详解

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

323. 无向图中的连通分量数目（中等）

130. 被围绕的区域（中等）

990. 等式方程的可满足性（中等）

今天讲讲 Union-Find 算法，也就是常说的并查集算法，主要是解决图论中「动态连通性」问题的。名词很高端，其实特别好理解，等会解释，另外这个算法的应用都非常有趣。

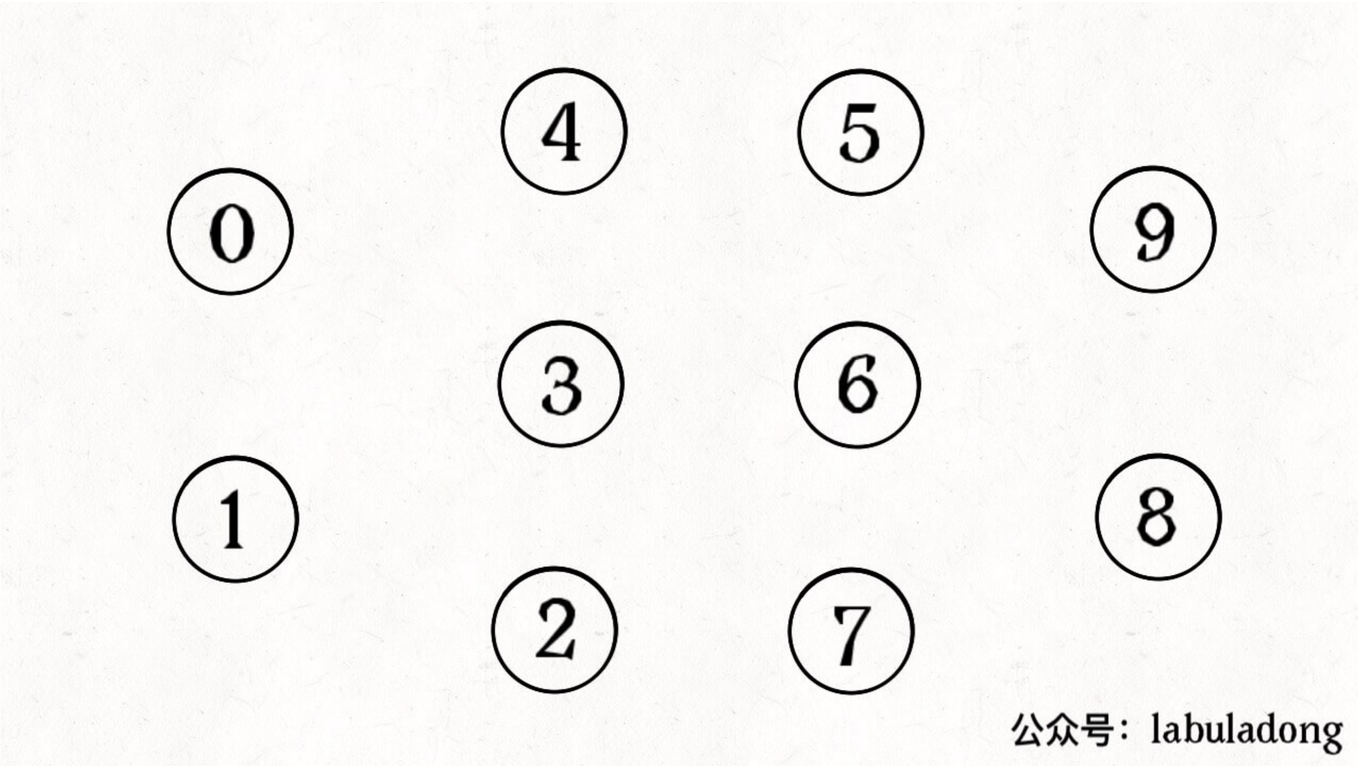
说起这个 Union-Find，应该算是我的「启蒙算法」了，因为《算法4》的开头就介绍了这款算法，可是把我秀翻了，感觉好精妙啊！

后来刷了 LeetCode，并查集相关的算法题目都非常有意思，而且《算法4》给的解法竟然还可以进一步优化，只要加一个微小的修改就可以把时间复杂度降到 O(1)。

废话不多说，直接上干货，先解释一下什么叫动态连通性吧。

一、问题介绍

简单说，动态连通性其实可以抽象成给一幅图连线。比如下面这幅图，总共有 10 个节点，他们互不相连，分别用 0~9 标记：



现在我们的 Union-Find 算法主要需要实现这两个 API：

```
class UF {
    /* 将 p 和 q 连接 */
    public void union(int p, int q);
    /* 判断 p 和 q 是否连通 */
    public boolean connected(int p, int q);
    /* 返回图中有多少个连通分量 */
    public int count();
}
```

这里所说的「连通」是一种等价关系，也就是说具有如下三个性质：

1、自反性：节点 `p` 和 `p` 是连通的。

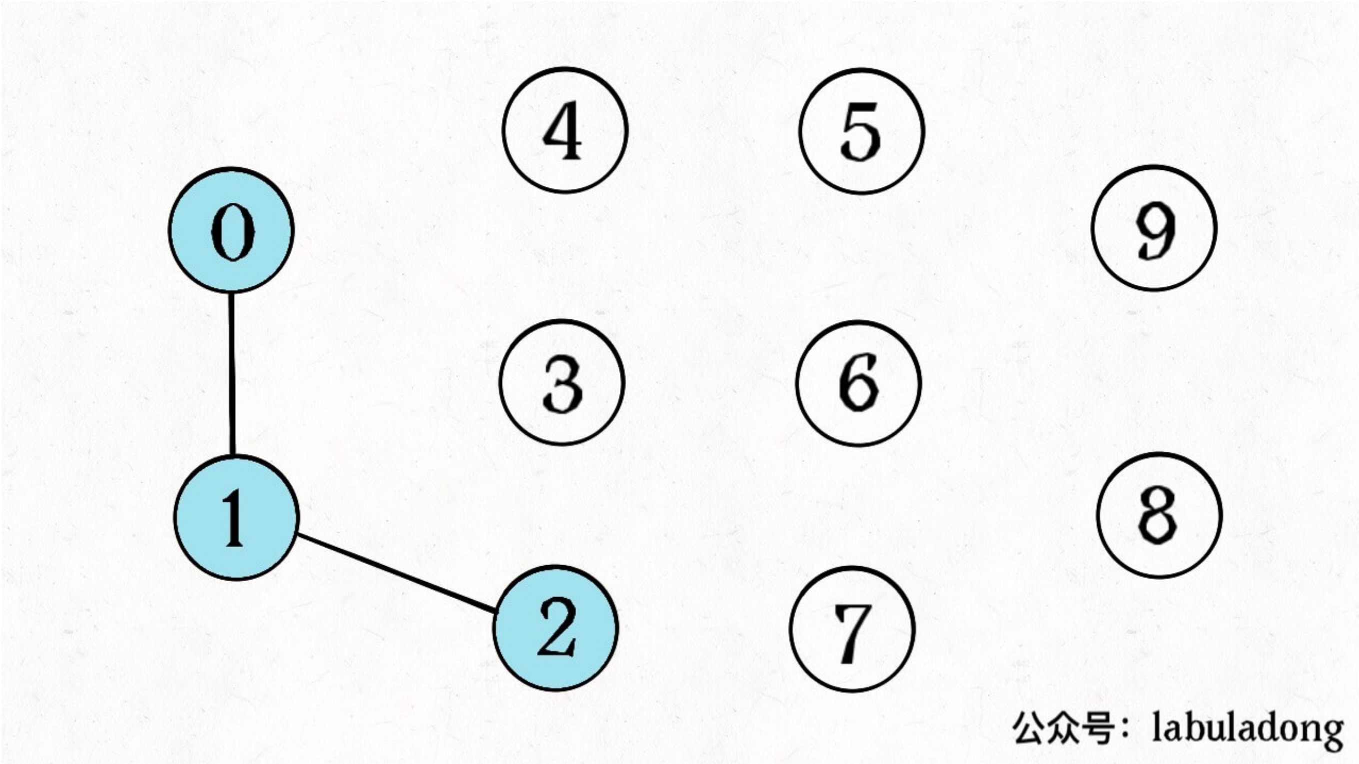
2、对称性：如果节点 `p` 和 `q` 连通，那么 `q` 和 `p` 也连通。

3、传递性：如果节点 `p` 和 `q` 连通，`q` 和 `r` 连通，那么 `p` 和 `r` 也连通。

比如说之前那幅图，0～9 任意两个**不同**的点都不连通，调用 `connected` 都会返回 false，连通分量为 10 个。

如果现在调用 `union(0, 1)`，那么 0 和 1 被连通，连通分量降为 9 个。

再调用 `union(1, 2)`，这时 0,1,2 都被连通，调用 `connected(0, 2)` 也会返回 true，连通分量变为 8 个。



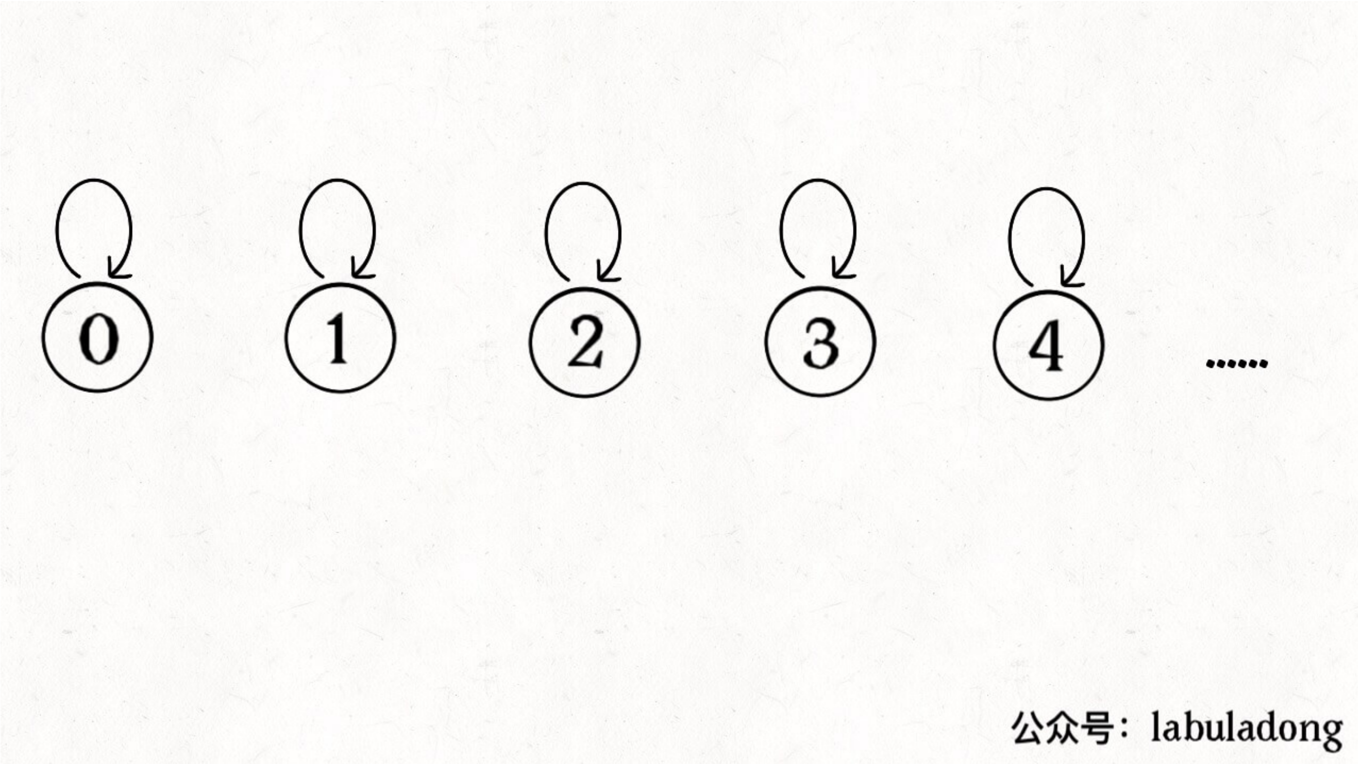
判断这种「等价关系」非常实用，比如说编译器判断同一个变量的不同引用，比如社交网络中的朋友圈计算等等。

这样，你应该大概明白什么是动态连通性了，Union-Find 算法的关键就在于 `union` 和 `connected` 函数的效率。那么用什么模型来表示这幅图的连通状态呢？用什么数据结构来实现代码呢？

二、基本思路

注意我刚才把「模型」和具体的「数据结构」分开说，这么做是有原因的。因为我们使用森林（若干棵树）来表示图的动态连通性，用数组来具体实现这个森林。

怎么用森林来表示连通性呢？我们设定树的每个节点有一个指针指向其父节点，如果是根节点的话，这个指针指向自己。比如说刚才那幅 10 个节点的图，一开始的时候没有相互连通，就是这样：

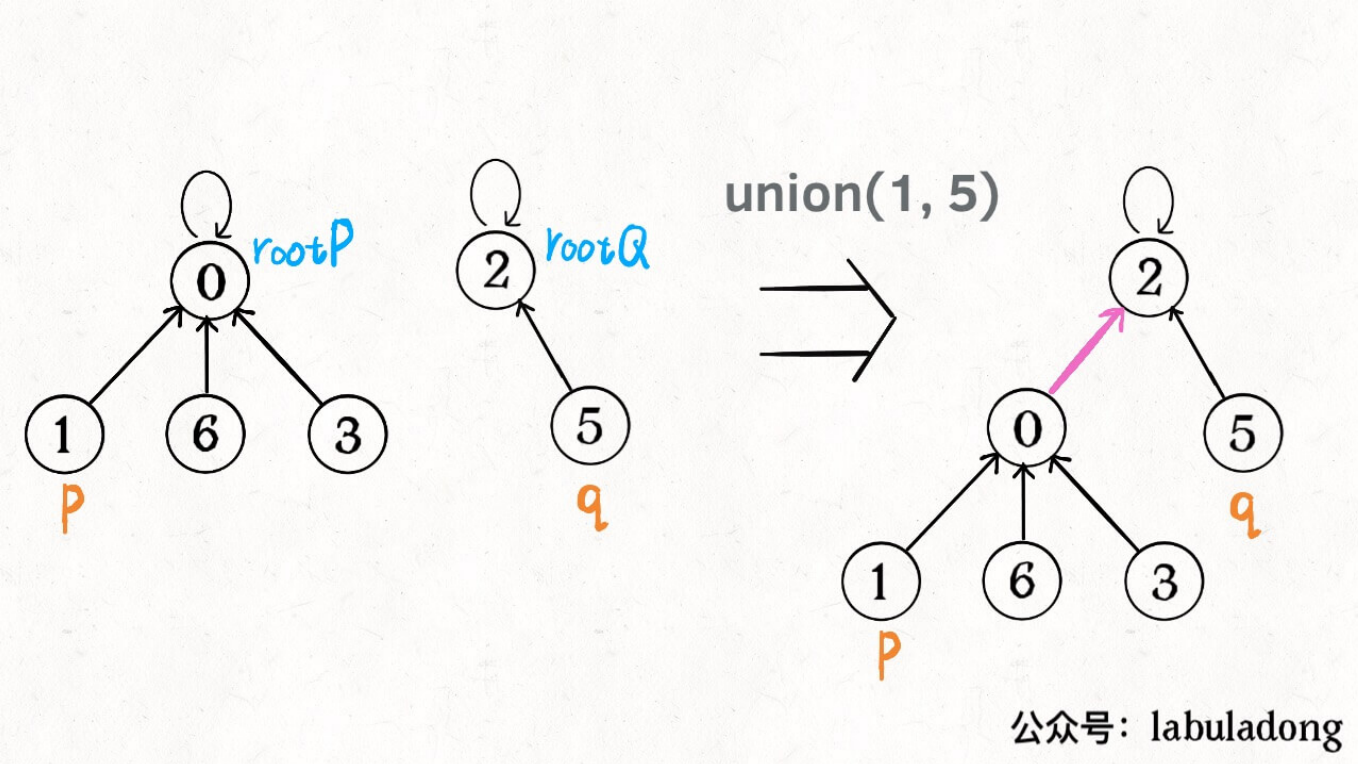



```java
class UF {
    // 记录连通分量
    private int count;
    // 节点 x 的节点是 parent[x]
    private int[] parent;

    /* 构造函数，n 为图的节点总数 */
    public UF(int n) {
        // 一开始互不连通
        this.count = n;
        // 父节点指针初始指向自己
        parent = new int[n];
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }

    /* 其他函数 */
}
```

如果某两个节点被连通，则让其中的（任意）一个节点的根节点接到另一个节点的根节点上：

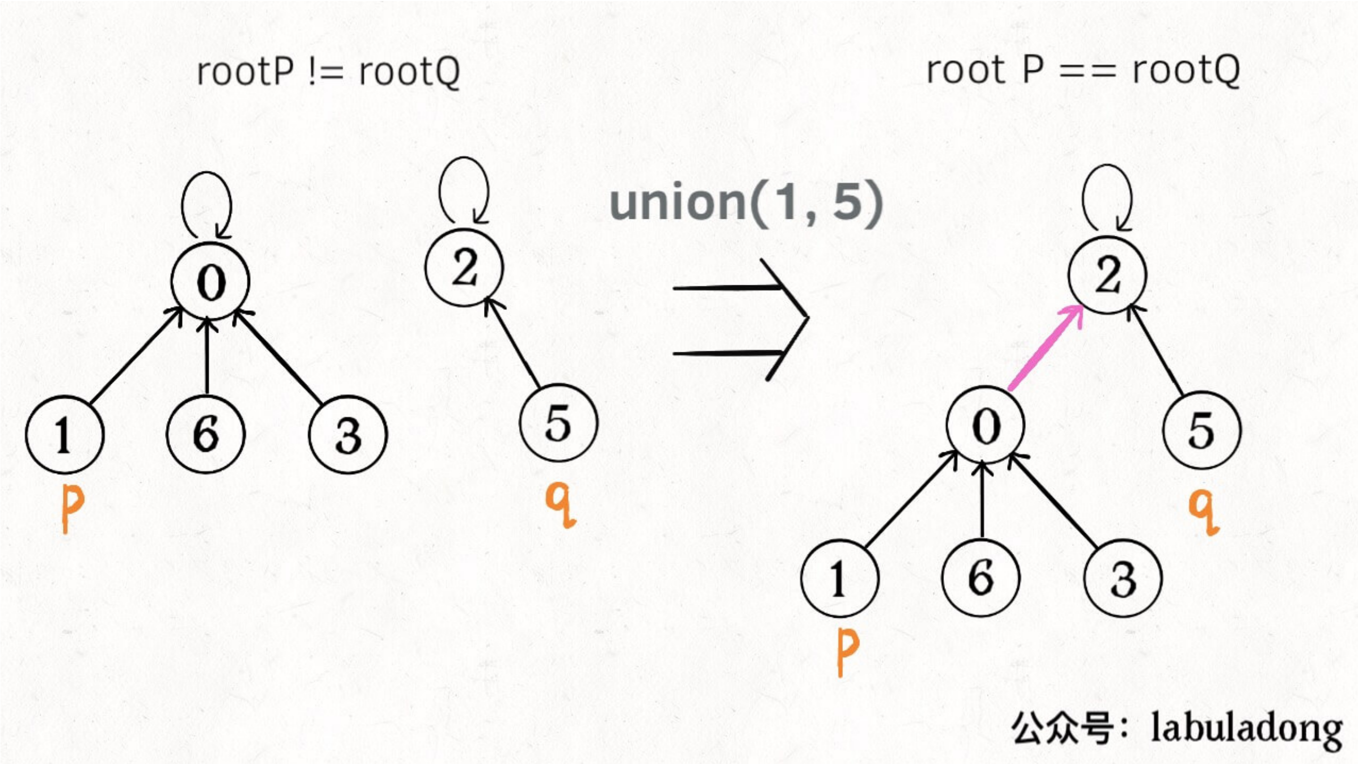



```
public void union(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if (rootP == rootQ)
        return;
    // 将两棵树合并为一棵
    parent[rootP] = rootQ;
    // parent[rootQ] = rootP 也一样
    count--; // 两个分量合二为一
}

/* 返回某个节点 x 的根节点 */
private int find(int x) {
    // 根节点的 parent[x] == x
    while (parent[x] != x)
        x = parent[x];
    return x;
}

/* 返回当前的连通分量个数 */
public int count() {
    return count;
}
```

这样，如果节点 p 和 q 连通的话，它们一定拥有相同的根节点：

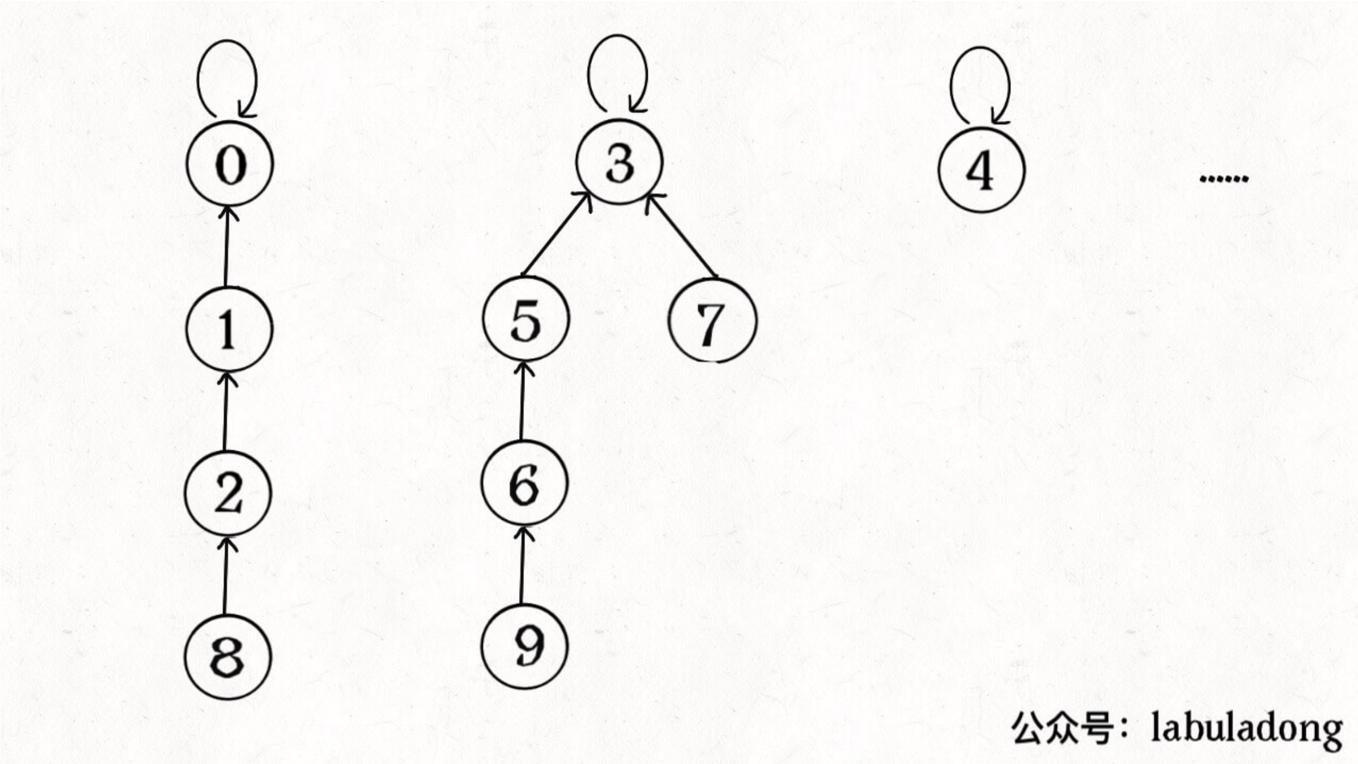



```
public boolean connected(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    return rootP == rootQ;
}
```

至此，Union-Find 算法就基本完成了。是不是很神奇？竟然可以这样使用数组来模拟出一个森林，如此巧妙的解决这个比较复杂的问题！

那么这个算法的复杂度是多少呢？我们发现，主要 API `connected` 和 `union` 中的复杂度都是 `find` 函数造成的，所以说它们的复杂度和 `find` 一样。

`find` 主要功能就是从某个节点向上遍历到树根，其时间复杂度就是树的高度。我们可能习惯性地认为树的高度就是 `logN`，但这并不一定。`logN` 的高度只存在于平衡二叉树，对于一般的树可能出现极端不平衡的情况，使得「树」几乎退化成「链表」，树的高度最坏情况下可能变成 `N`。



所以说上面这种解法，`find`，`union`，`connected` 的时间复杂度都是 O(N)。这个复杂度很不理想的，你想图论解决的都是诸如社交网络这样数据规模巨大的问题，对于 `union` 和 `connected` 的调用非常频繁，每次调用需要线性时间完全不可忍受。

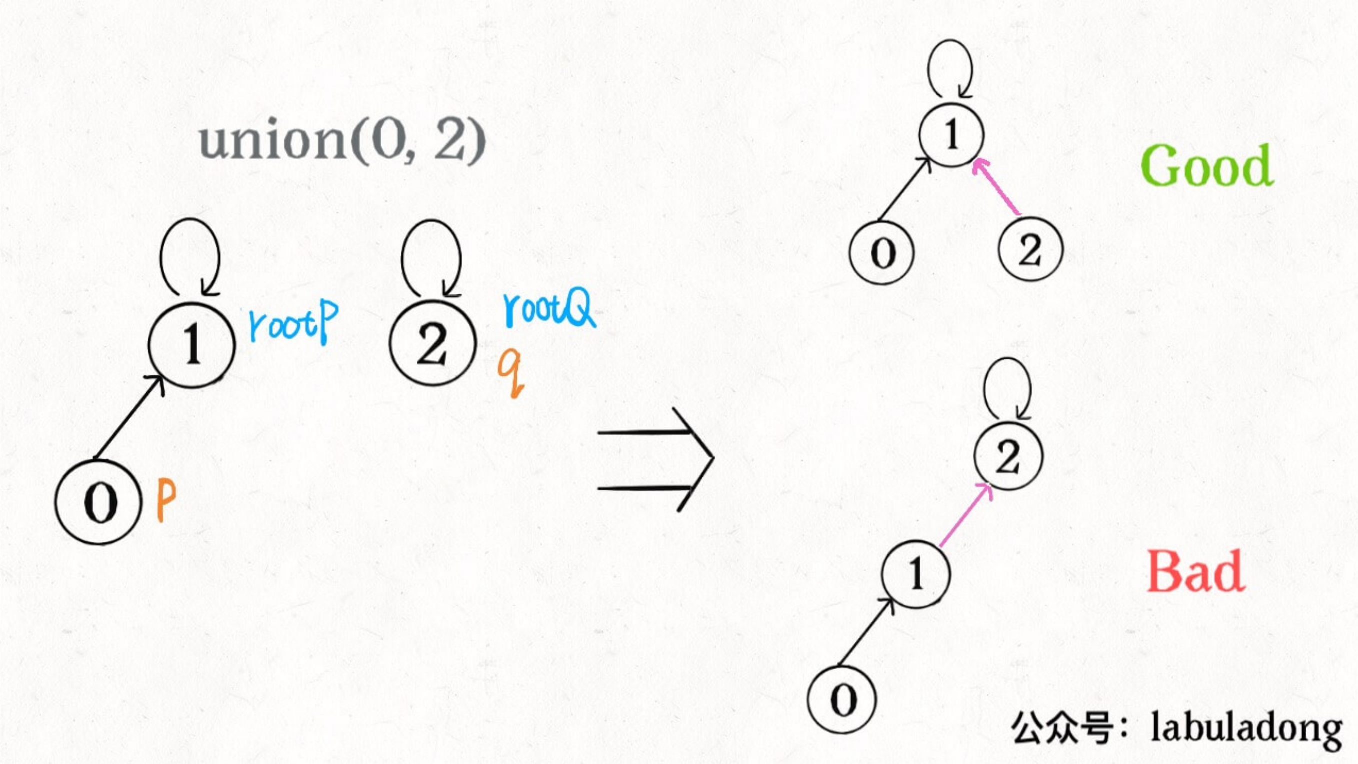
问题的关键在于，如何想办法避免树的不平衡呢？只需要略施小计即可。

三、平衡性优化

我们要知道哪种情况下可能出现不平衡现象，关键在于 `union` 过程：

```
public void union(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if (rootP == rootQ)
        return;
    // 将两棵树合并为一棵
    parent[rootP] = rootQ;
    // parent[rootQ] = rootP 也可以
    count--;
```

我们一开始就是简单粗暴的把 `p` 所在的树接到 `q` 所在的树的根节点下面，那么这里就可能出现「头重脚轻」的不平衡状况，比如下面这种局面：



长此以往，树可能生长得很不平衡。**我们其实是希望，小一些的树接到大一些的树下面，这样就能避免头重脚轻，更平衡一些**。解决方法是额外使用一个 `size` 数组，记录每棵树包含的节点数，我们不妨称为「重量」：

```java
class UF {
    private int count;
    private int[] parent;
    // 新增一个数组记录树的“重量”
    private int[] size;

    public UF(int n) {
        this.count = n;
        parent = new int[n];
        // 最初每棵树只有一个节点
        // 重量应该初始化 1
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }
    /* 其他函数 */
}
```

比如说 `size[3] = 5` 表示，以节点 `3` 为根的那棵树，总共有 `5` 个节点。这样我们可以修改一下 `union` 方法：

```java
public void union(int p, int q) {
    int rootP = find(p);
```

```
    int rootQ = find(q);
    if (rootP == rootQ)
        return;

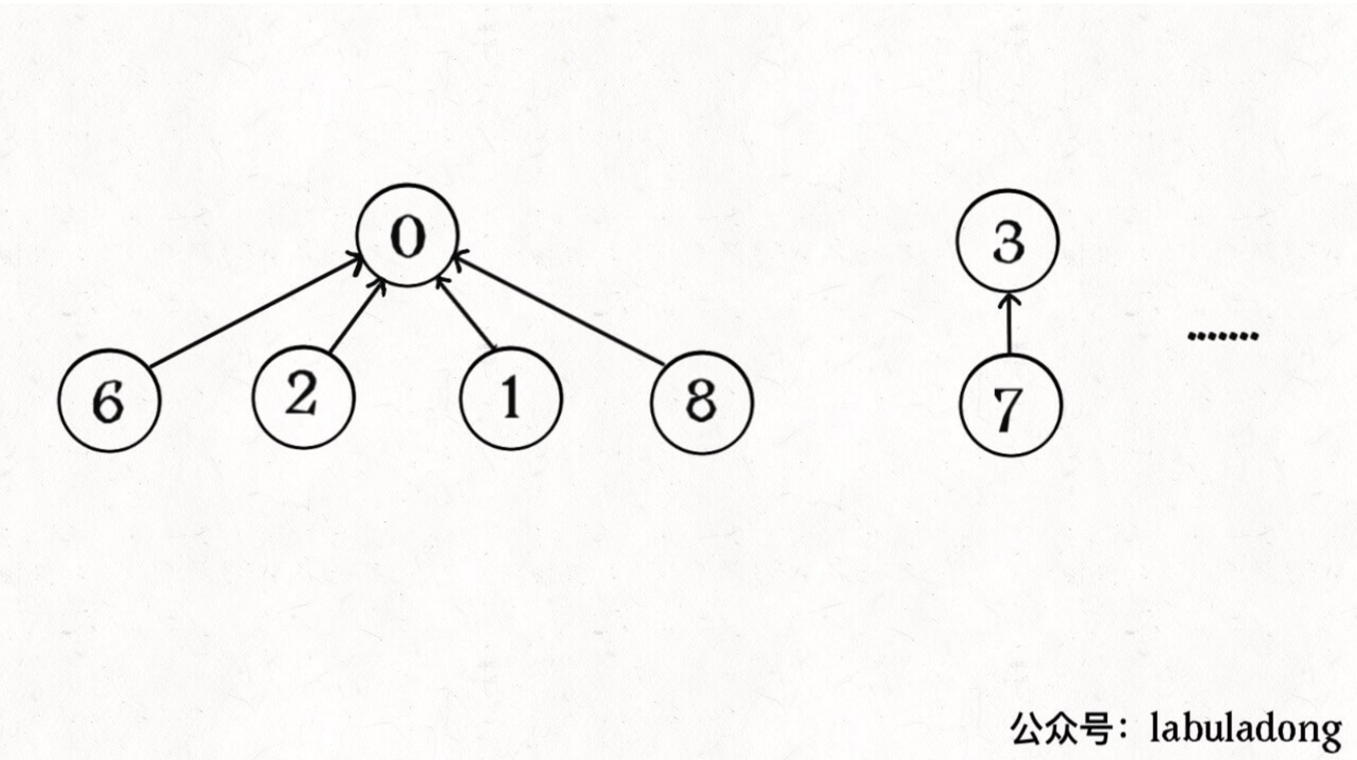
    // 小树接到大树下面，较平衡
    if (size[rootP] > size[rootQ]) {
        parent[rootQ] = rootP;
        size[rootP] += size[rootQ];
    } else {
        parent[rootP] = rootQ;
        size[rootQ] += size[rootP];
    }
    count--;
}
```

这样，通过比较树的重量，就可以保证树的生长相对平衡，树的高度大致在 `logN` 这个数量级，极大提升执行效率。

此时，`find`，`union`，`connected` 的时间复杂度都下降为 O(logN)，即便数据规模上亿，所需时间也非常少。

四、路径压缩

这步优化特别简单，所以非常巧妙。我们能不能进一步压缩每棵树的高度，使树高始终保持为常数？

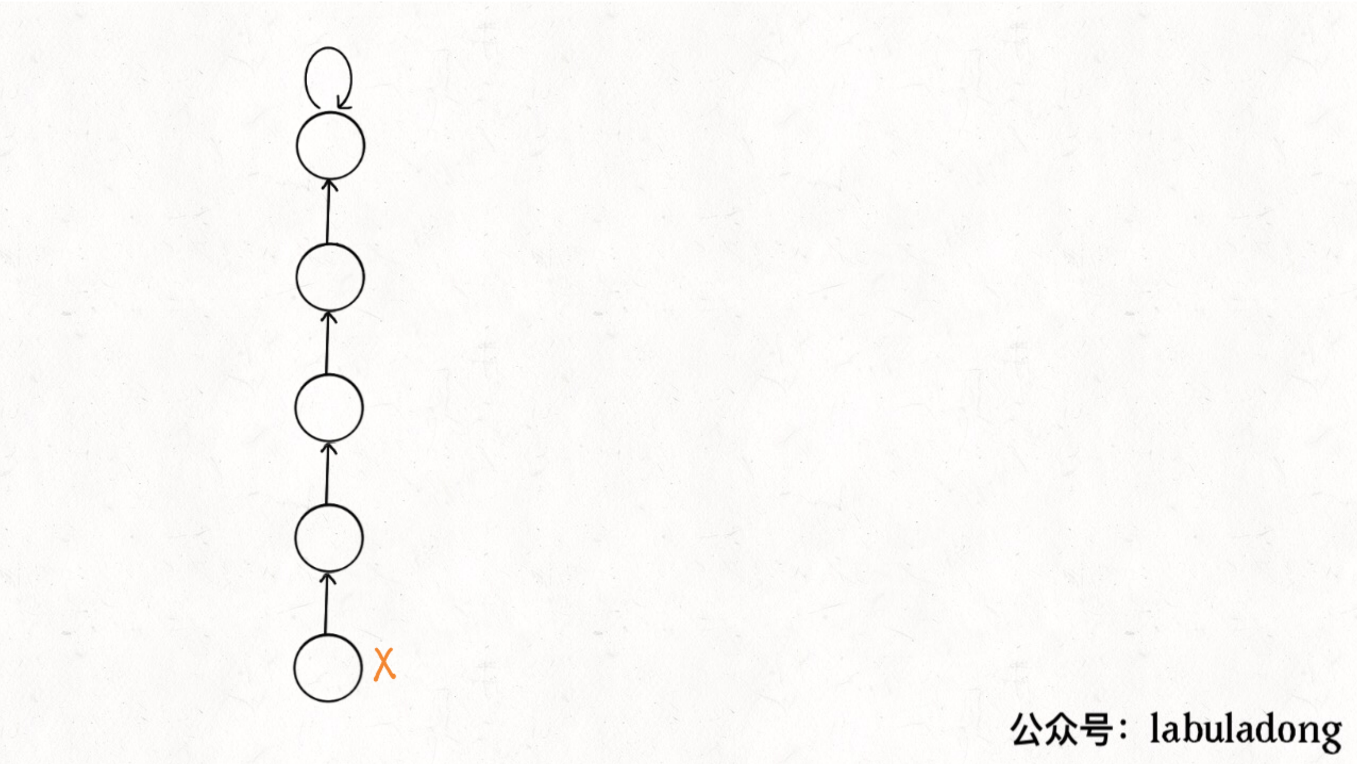


这样 `find` 就能以 O(1) 的时间找到某一节点的根节点，相应的，`connected` 和 `union` 复杂度都下降为 O(1)。

要做到这一点，非常简单，只需要在 `find` 中加一行代码：

```
private int find(int x) {
    while (parent[x] != x) {
        // 进行路径压缩
        parent[x] = parent[parent[x]];
        x = parent[x];
    }
    return x;
}
```

这个操作有点匪夷所思，看个 GIF 就明白它的作用了（为清晰起见，这棵树比较极端）：



可见，调用 find 函数每次向树根遍历的同时，顺手将树高缩短了，最终所有树高都不会超过 3（union 的时候树高可能达到 3），树高为常数，那么所有方法的复杂度也就都是 O(1)。

> PS：读者可能会问，这个 GIF 图的 find 过程完成之后，树高恰好等于 3 了，但是如果更高的树，压缩后高度依然会大于 3 呀？不能这么想。这个 GIF 的情景是我编出来方便大家理解路径压缩的，但是实际中，每次 find 都会进行路径压缩，所以树本来就不可能增长到这么高，你的这种担心应该是多余的。

当然，如果路径压缩技巧将树高保持为常数了，那么 size 数组的平衡化优化就不是特别必要了。

你一般看到的 Union Find 算法应该是使用路径压缩的实现：

```
class UF {
    // 连通分量个数
    private int count;
    // 存储每个节点的父节点
    private int[] parent;
```

```java
    // n 为图中节点的个数
    public UF(int n) {
        this.count = n;
        parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    // 将节点 p 和节点 q 连通
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);

        if (rootP == rootQ)
            return;

        parent[rootQ] = rootP;
        // 两个连通分量合并成一个连通分量
        count--;
    }

    // 判断节点 p 和节点 q 是否连通
    public boolean connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

    // 返回节点 x 的连通分量根节点
    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    // 返回图中的连通分量个数
    public int count() {
        return count;
    }
}
```

Union-Find 算法的复杂度可以这样分析：构造函数初始化数据结构需要 O(N) 的时间和空间复杂度；连通两个节点 `union`、判断两个节点的连通性 `connected`、计算连通分量 `count` 所需的时间复杂度均为 O(1)。

到这里，相信你已经掌握了 Union-Find 算法的核心逻辑，总结一下我们优化算法的过程：

1、用 `parent` 数组记录每个节点的父节点，相当于指向父节点的指针，所以 `parent` 数组内实际存储着一个森林（若干棵多叉树）。

2、用 size 数组记录着每棵树的重量，目的是让 union 后树依然拥有平衡性，保证各个 API 时间复杂度为 O(logN)，而不会退化成链表影响操作效率。

3、在 find 函数中进行路径压缩，保证任意树的高度保持在常数，使得各个 API 时间复杂度为 O(1)。使用了路径压缩之后，可以不使用 size 数组的平衡优化。

下面我们看一些具体的并查集题目。

题目实践

力扣第 323 题「无向图中连通分量的数目」就是最基本的连通分量题目：

给你输入一个包含 n 个节点的图，用一个整数 n 和一个数组 edges 表示，其中 edges[i] = [ai, bi] 表示图中节点 ai 和 bi 之间有一条边。请你计算这幅图的连通分量个数。

函数签名如下：

```
int countComponents(int n, int[][] edges)
```

这道题我们可以直接套用 UF 类来解决：

```
public int countComponents(int n, int[][] edges) {
    UF uf = new UF(n);
    // 将每个节点进行连通
    for (int[] e : edges) {
        uf.union(e[0], e[1]);
    }
    // 返回连通分量的个数
    return uf.count();
}

class UF {
    // 见上文
}
```

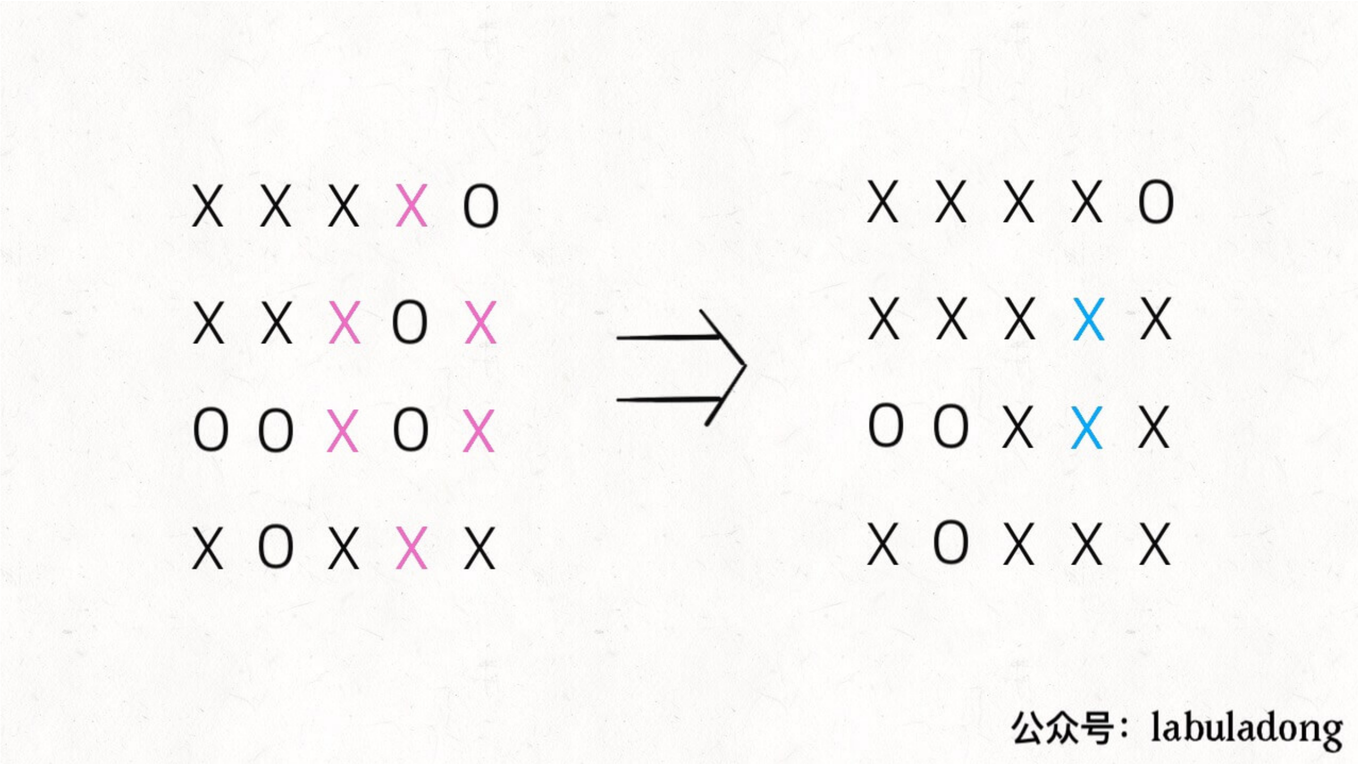
另外，一些使用 DFS 深度优先算法解决的问题，也可以用 Union-Find 算法解决。

比如力扣第 130 题「被围绕的区域」：

给你一个 M×N 的二维矩阵，其中包含字符 X 和 O，让你找到矩阵中**四面**被 X 围住的 O，并且把它们替换成 X。

```
void solve(char[][] board);
```

注意哦，必须是四面被围的 O 才能被换成 X，也就是说边角上的 O 一定不会被围，进一步，与边角上的 O 相连的 O 也不会被 X 围四面，也不会被替换。



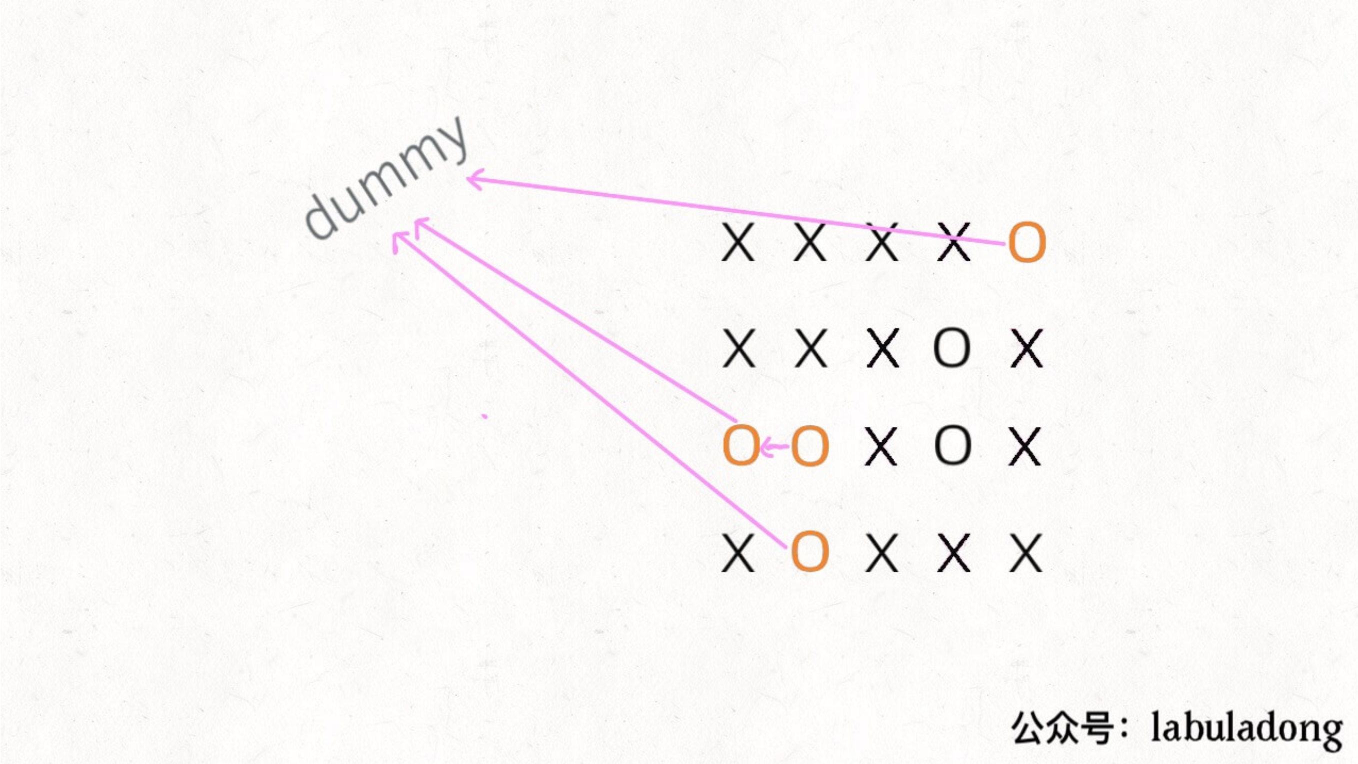
> PS：这让我想起小时候玩的棋类游戏「黑白棋」，只要你用两个棋子把对方的棋子夹在中间，对方的子就被替换成你的子。可见，占据四角的棋子是无敌的，与其相连的边棋子也是无敌的（无法被夹掉）。

其实这个问题应该归为 岛屿系列问题 使用 DFS 算法解决：

先用 for 循环遍历棋盘的**四边**，用 DFS 算法把那些与边界相连的 `O` 换成一个特殊字符，比如 `#`；然后再遍历整个棋盘，把剩下的 `O` 换成 `X`，把 `#` 恢复成 `O`。这样就能完成题目的要求，时间复杂度 O(MN)。

但这个问题也可以用 Union-Find 算法解决，虽然实现复杂一些，甚至效率也略低，但这是使用 Union-Find 算法的通用思想，值得一学。

你可以把那些不需要被替换的 `O` 看成一个拥有独门绝技的门派，它们有一个共同「祖师爷」叫 `dummy`，这些 `O` 和 `dummy` 互相连通，而那些需要被替换的 `O` 与 `dummy` 不连通。



这就是 Union-Find 的核心思路，明白这个图，就很容易看懂代码了。

首先要解决的是，根据我们的实现，Union-Find 底层用的是一维数组，构造函数需要传入这个数组的大小，而题目给的是一个二维棋盘。

这个很简单，二维坐标 `(x,y)` 可以转换成 `x * n + y` 这个数（`m` 是棋盘的行数，`n` 是棋盘的列数），**敲黑板，这是将二维坐标映射到一维的常用技巧。**

其次，我们之前描述的「祖师爷」是虚构的，需要给他老人家留个位置。索引 `[0.. m*n-1]` 都是棋盘内坐标的一维映射，那就让这个虚拟的 `dummy` 节点占据索引 `m * n` 好了。

看解法代码：

```
void solve(char[][] board) {
    if (board.length == 0) return;

    int m = board.length;
    int n = board[0].length;
    // 给 dummy 留一个额外位置
    UF uf = new UF(m * n + 1);
    int dummy = m * n;
    // 将首列和末列的 O 与 dummy 连通
    for (int i = 0; i < m; i++) {
        if (board[i][0] == 'O')
            uf.union(i * n, dummy);
        if (board[i][n - 1] == 'O')
            uf.union(i * n + n - 1, dummy);
    }
    // 将首行和末行的 O 与 dummy 连通
    for (int j = 0; j < n; j++) {
        if (board[0][j] == 'O')
```

```java
                uf.union(j, dummy);
            if (board[m - 1][j] == 'O')
                uf.union(n * (m - 1) + j, dummy);
        }
        // 方向数组 d 是上下左右搜索的常用手法
        int[][] d = new int[][]{{1,0}, {0,1}, {0,-1}, {-1,0}};
        for (int i = 1; i < m - 1; i++)
            for (int j = 1; j < n - 1; j++)
                if (board[i][j] == 'O')
                    // 将此 O 与上下左右的 O 连通
                    for (int k = 0; k < 4; k++) {
                        int x = i + d[k][0];
                        int y = j + d[k][1];
                        if (board[x][y] == 'O')
                            uf.union(x * n + y, i * n + j);
                    }
        // 所有不和 dummy 连通的 O，都要被替换
        for (int i = 1; i < m - 1; i++)
            for (int j = 1; j < n - 1; j++)
                if (!uf.connected(dummy, i * n + j))
                    board[i][j] = 'X';
    }

    class UF {
        // 见上文
    }
```

这段代码很长，其实就是刚才的思路实现，只有和边界 O 相连的 O 才具有和 dummy 的连通性，他们不会被替换。

其实用 Union-Find 算法解决这个简单的问题有点杀鸡用牛刀，它可以解决更复杂，更具有技巧性的问题，**主要思路是适时增加虚拟节点，想办法让元素「分门别类」，建立动态连通关系**。

力扣第 990 题「等式方程的可满足性」用 Union-Find 算法就显得十分优美了，题目是这样：

给你一个数组 equations，装着若干字符串表示的算式。每个算式 equations[i] 长度都是 4，而且只有这两种情况：a==b 或者 a!=b，其中 a,b 可以是任意小写字母。你写一个算法，如果 equations 中所有算式都不会互相冲突，返回 true，否则返回 false。

比如说，输入 ["a==b","b!=c","c==a"]，算法返回 false，因为这三个算式不可能同时正确。

再比如，输入 ["c==c","b==d","x!=z"]，算法返回 true，因为这三个算式并不会造成逻辑冲突。

我们前文说过，动态连通性其实就是一种等价关系，具有「自反性」「传递性」和「对称性」，其实 == 关系也是一种等价关系，具有这些性质。所以这个问题用 Union-Find 算法就很自然。

核心思想是，将 equations 中的算式根据 == 和 != 分成两部分，先处理 == 算式，使得他们通过相等关系各自勾结成门派（连通分量）；然后处理 != 算式，检查不等关系是否破坏了相等关系的连通性。

```java
boolean equationsPossible(String[] equations) {
    // 26 个英文字母
```

```java
    UF uf = new UF(26);
    // 先让相等的字母形成连通分量
    for (String eq : equations) {
        if (eq.charAt(1) == '=') {
            char x = eq.charAt(0);
            char y = eq.charAt(3);
            uf.union(x - 'a', y - 'a');
        }
    }
    // 检查不等关系是否打破相等关系的连通性
    for (String eq : equations) {
        if (eq.charAt(1) == '!') {
            char x = eq.charAt(0);
            char y = eq.charAt(3);
            // 如果相等关系成立，就是逻辑冲突
            if (uf.connected(x - 'a', y - 'a'))
                return false;
        }
    }
    return true;
}

class UF {
    // 见上文
}
```

至此，这道判断算式合法性的问题就解决了，借助 Union-Find 算法，是不是很简单呢？

最后，Union-Find 算法也会在一些其他经典图论算法中用到，比如判断「图」和「树」，以及最小生成树的计算，详情见 Kruskal 最小生成树算法。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜

labuladong公众号

Kruskal 最小生成树算法

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

261. 以图判树（中等）

1135. 最低成本联通所有城市（中等）

1584. 连接所有点的最小费用（中等）

图论中知名度比较高的算法应该就是 Dijkstra 最短路径算法，环检测和拓扑排序，二分图判定算法 以及今天要讲的最小生成树（Minimum Spanning Tree）算法了。

最小生成树算法主要有 Prim 算法（普里姆算法）和 Kruskal 算法（克鲁斯卡尔算法）两种，这两种算法虽然都运用了贪心思想，但从实现上来说差异还是蛮大的。

本文先来讲比较简单易懂的 Kruskal 算法，然后在下一篇文章 Prim 算法模板 中聊 Prim 算法。

Kruskal 算法其实很容易理解和记忆，其关键是要熟悉并查集算法，如果不熟悉，建议先看下前文 Union-Find 并查集算法。

接下来，我们从最小生成树的定义说起。

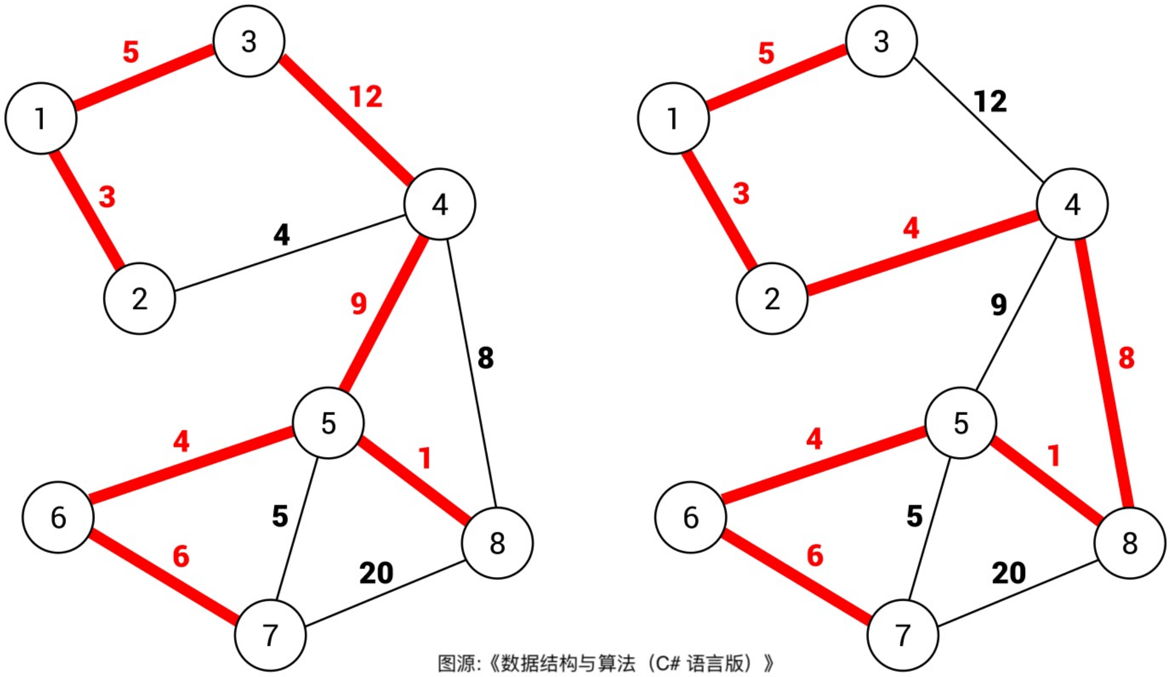
什么是最小生成树

先说「树」和「图」的根本区别：树不会包含环，图可以包含环。

如果一幅图没有环，完全可以拉伸成一棵树的模样。说的专业一点，树就是「无环连通图」。

那么什么是图的「生成树」呢，其实按字面意思也好理解，就是在图中找一棵包含图中的所有节点的树。专业点说，生成树是含有图中所有顶点的「无环连通子图」。

容易想到，一幅图可以有很多不同的生成树，比如下面这幅图，红色的边就组成了两棵不同的生成树：



对于加权图，每条边都有权重，所以每棵生成树都有一个权重和。比如上图，右侧生成树的权重和显然比左侧生成树的权重和要小。

那么最小生成树很好理解了，所有可能的生成树中，权重和最小的那棵生成树就叫「最小生成树」。

> PS：一般来说，我们都是在**无向加权图**中计算最小生成树的，所以使用最小生成树算法的现实场景中，图的边权重一般代表成本、距离这样的标量。

在讲 Kruskal 算法之前，需要回顾一下 Union-Find 并查集算法。

Union-Find 并查集算法

刚才说了，图的生成树是含有其所有顶点的「无环连通子图」，最小生成树是权重和最小的生成树。

那么说到连通性，相信老读者应该可以想到 Union-Find 并查集算法，用来高效处理图中联通分量的问题。

前文 Union-Find 并查集算法详解 详细介绍了 Union-Find 算法的实现原理，主要运用 `size` 数组和路径压缩技巧提高连通分量的判断效率。

如果不了解 Union-Find 算法的读者可以去看前文，为了节约篇幅，本文直接给出 Union-Find 算法的实现：

```
class UF {
    // 连通分量个数
    private int count;
    // 存储一棵树
    private int[] parent;
    // 记录树的「重量」
    private int[] size;

    // n 为图中节点的个数
    public UF(int n) {
        this.count = n;
```

```java
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    // 将节点 p 和节点 q 连通
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;

        // 小树接到大树下面，较平衡
        if (size[rootP] > size[rootQ]) {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        } else {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        }
        // 两个连通分量合并成一个连通分量
        count--;
    }

    // 判断节点 p 和节点 q 是否连通
    public boolean connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

    // 返回节点 x 的连通分量根节点
    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    // 返回图中的连通分量个数
    public int count() {
        return count;
    }
}
```

前文 Union-Find 并查集算法详解 介绍过 Union-Find 算法的一些算法场景，而它在 Kruskal 算法中的主要作用是保证最小生成树的合法性。

因为在构造最小生成树的过程中，你首先得保证生成的那玩意是棵树（不包含环）对吧，那么 Union-Find 算法就是帮你干这个事儿的。

怎么做到的呢？先来看看力扣第 261 题「以图判树」，我描述下题目：

给你输入编号从 `0` 到 `n - 1` 的 `n` 个结点，和一个无向边列表 `edges`（每条边用节点二元组表示），请你判断输入的这些边组成的结构是否是一棵树。

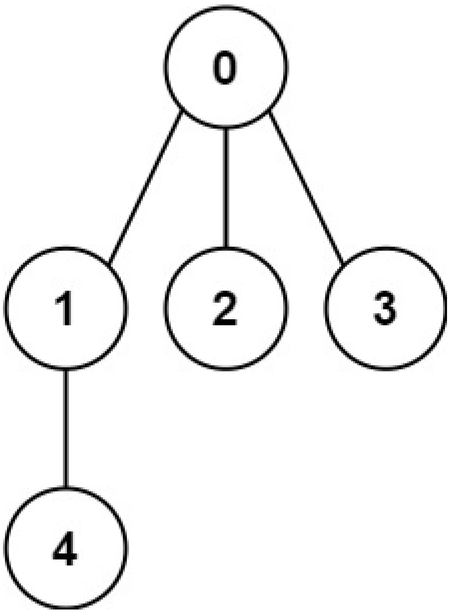
函数签名如下：

```
boolean validTree(int n, int[][] edges);
```

比如输入如下：

```
n = 5
edges = [[0,1], [0,2], [0,3], [1,4]]
```

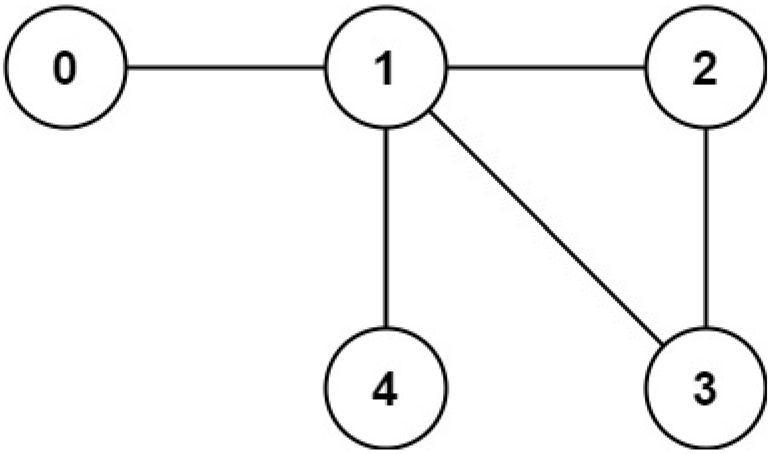
这些边构成的是一棵树，算法应该返回 true：



但如果输入：

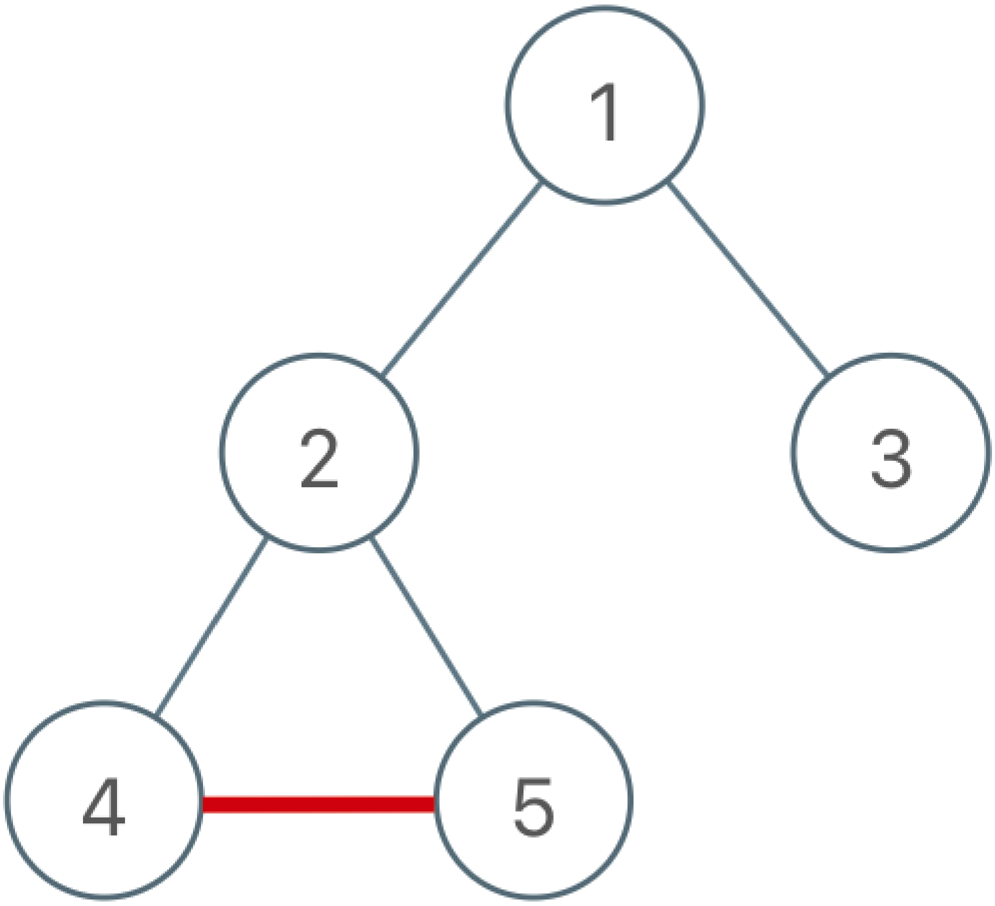
```
n = 5
edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]
```

形成的就不是树结构了，因为包含环：

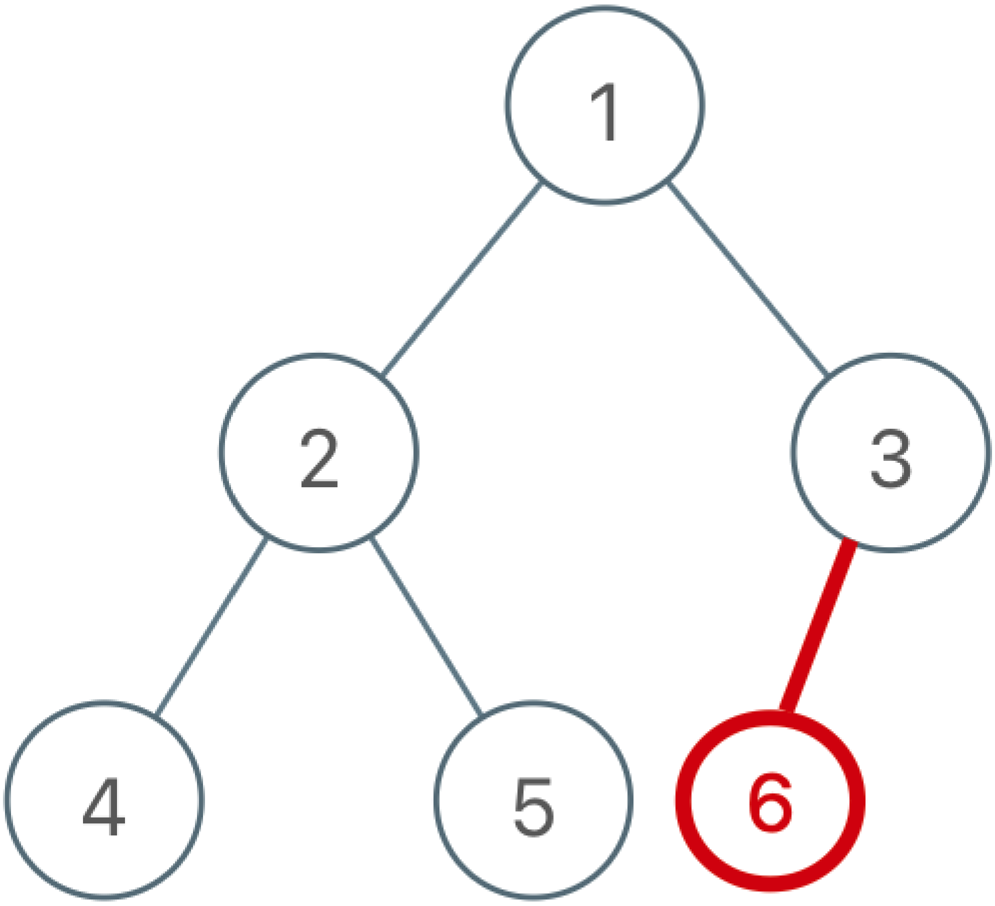


对于这道题，我们可以思考一下，什么情况下加入一条边会使得树变成图（出现环）？

显然，像下面这样添加边会出现环：



而这样添加边则不会出现环：



总结一下规律就是：

对于添加的这条边，如果该边的两个节点本来就在同一连通分量里，那么添加这条边会产生环；反之，如果该边的两个节点不在同一连通分量里，则添加这条边不会产生环。

而判断两个节点是否连通（是否在同一个连通分量中）就是 Union-Find 算法的拿手绝活，所以这道题的解法代码如下：

```
// 判断输入的若干条边是否能构造出一棵树结构
boolean validTree(int n, int[][] edges) {
    // 初始化 0...n-1 共 n 个节点
    UF uf = new UF(n);
    // 遍历所有边，将组成边的两个节点进行连接
    for (int[] edge : edges) {
        int u = edge[0];
        int v = edge[1];
        // 若两个节点已经在同一连通分量中，会产生环
        if (uf.connected(u, v)) {
            return false;
        }
```

```
        // 这条边不会产生环，可以是树的一部分
        uf.union(u, v);
    }
    // 要保证最后只形成了一棵树，即只有一个连通分量
    return uf.count() == 1;
}

class UF {
    // 见上文代码实现
}
```

如果你能够看懂这道题的解法思路，那么掌握 Kruskal 算法就很简单了。

Kruskal 算法

所谓最小生成树，就是图中若干边的集合（我们后文称这个集合为 `mst`，最小生成树的英文缩写），你要保证这些边：

1、包含图中的所有节点。

2、形成的结构是树结构（即不存在环）。

3、权重和最小。

有之前题目的铺垫，前两条其实可以很容易地利用 Union-Find 算法做到，关键在于第 3 点，如何保证得到的这棵生成树是权重和最小的。

应合作方要求，本文不便在此发布，请扫码关注回复关键词「生成树」查看：

我写了个模板，把 Dijkstra 算法变成了默写题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

743. 网络延迟时间（中等）

1514. 概率最大的路径（中等）

1631. 最小体力消耗路径（中等）

其实，很多算法的底层原理异常简单，无非就是一步一步延伸，变得**看起来**好像特别复杂，特别牛逼。

但如果你看过历史文章，应该可以对算法形成自己的理解，就会发现很多算法都是换汤不换药，毫无新意，非常枯燥。

比如，我们说二叉树非常重要，你把这个结构掌握了，就会发现 动态规划，分治算法，回溯（DFS）算法，BFS 算法框架，Union-Find 并查集算法，二叉堆实现优先级队列 就是把二叉树翻来覆去的运用。

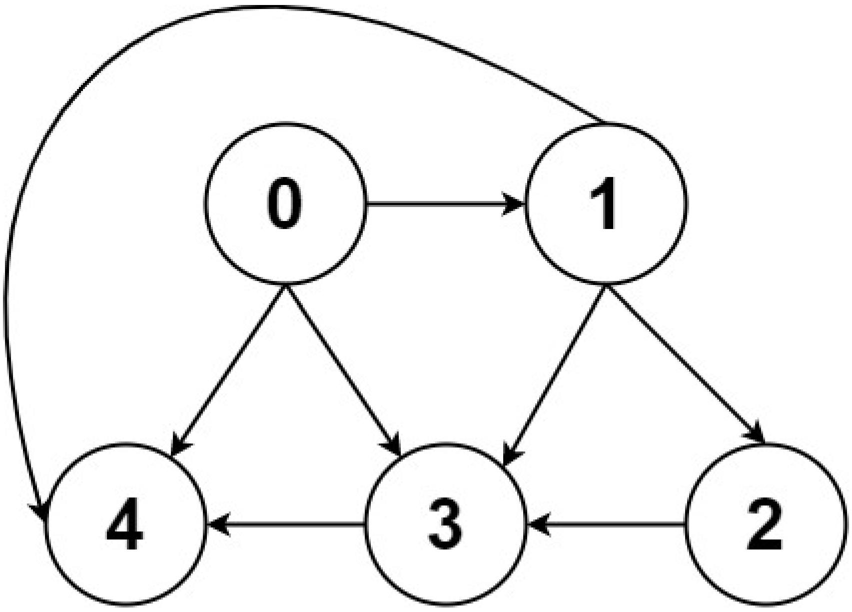
那么本文又要告诉你，Dijkstra 算法（一般音译成迪杰斯特拉算法）无非就是一个 BFS 算法的加强版，它们都是从二叉树的层序遍历衍生出来的。

这也是为什么我在 学习数据结构和算法的框架思维 中这么强调二叉树的原因。

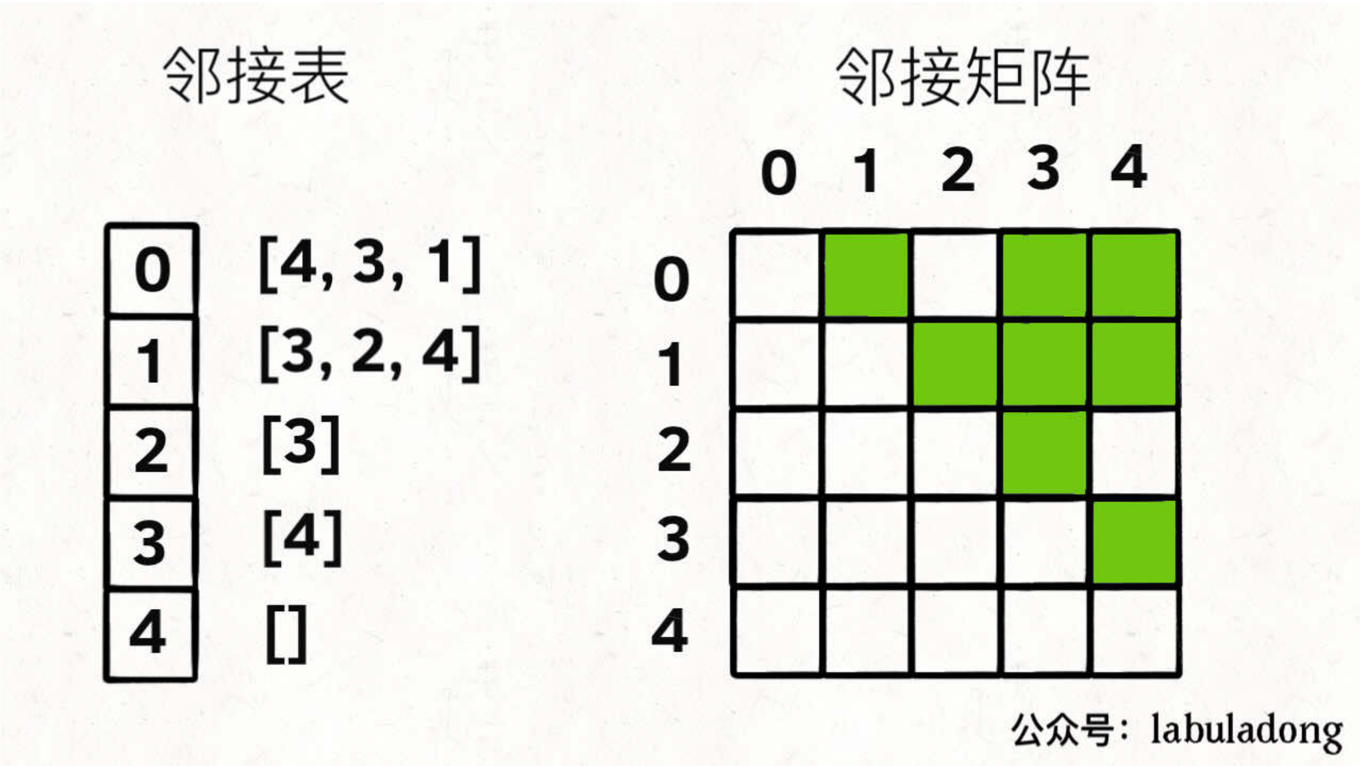
下面我们由浅入深，从二叉树的层序遍历聊到 Dijkstra 算法，给出 Dijkstra 算法的代码框架，顺手秒杀几道运用 Dijkstra 算法的题目。

图的抽象

前文 图论第一期：遍历基础 说过「图」这种数据结构的基本实现，图中的节点一般就抽象成一个数字（索引），图的具体实现一般是「邻接矩阵」或者「邻接表」。



比如上图这幅图用邻接表和邻接矩阵的存储方式如下：



前文 图论第二期：拓扑排序 告诉你，我们用邻接表的场景更多，结合上图，一幅图可以用如下 Java 代码表示：

```
// graph[s] 存储节点 s 指向的节点（出度）
List<Integer>[] graph;
```

如果你想把一个问题抽象成「图」的问题，那么首先要实现一个 API `adj`：

```
// 输入节点 s 返回 s 的相邻节点
List<Integer> adj(int s);
```

类似多叉树节点中的 `children` 字段记录当前节点的所有子节点，`adj(s)` 就是计算一个节点 `s` 的相邻节点。

比如上面说的用邻接表表示「图」的方式，`adj` 函数就可以这样表示：

```
List<Integer>[] graph;

// 输入节点 s，返回 s 的相邻节点
List<Integer> adj(int s) {
    return graph[s];
}
```

当然，对于「加权图」，我们需要知道两个节点之间的边权重是多少，所以还可以抽象出一个 `weight` 方法：

```
// 返回节点 from 到节点 to 之间的边的权重
int weight(int from, int to);
```

这个 `weight` 方法可以根据实际情况而定，因为不同的算法题，题目给的「权重」含义可能不一样，我们存储权重的方式也不一样。

有了上述基础知识，就可以搞定 Dijkstra 算法了，下面我给你从二叉树的层序遍历开始推演出 Dijkstra 算法的实现。

二叉树层级遍历和 BFS 算法

我们之前说过二叉树的层级遍历框架：

```
// 输入一棵二叉树的根节点，层序遍历这棵二叉树
void levelTraverse(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

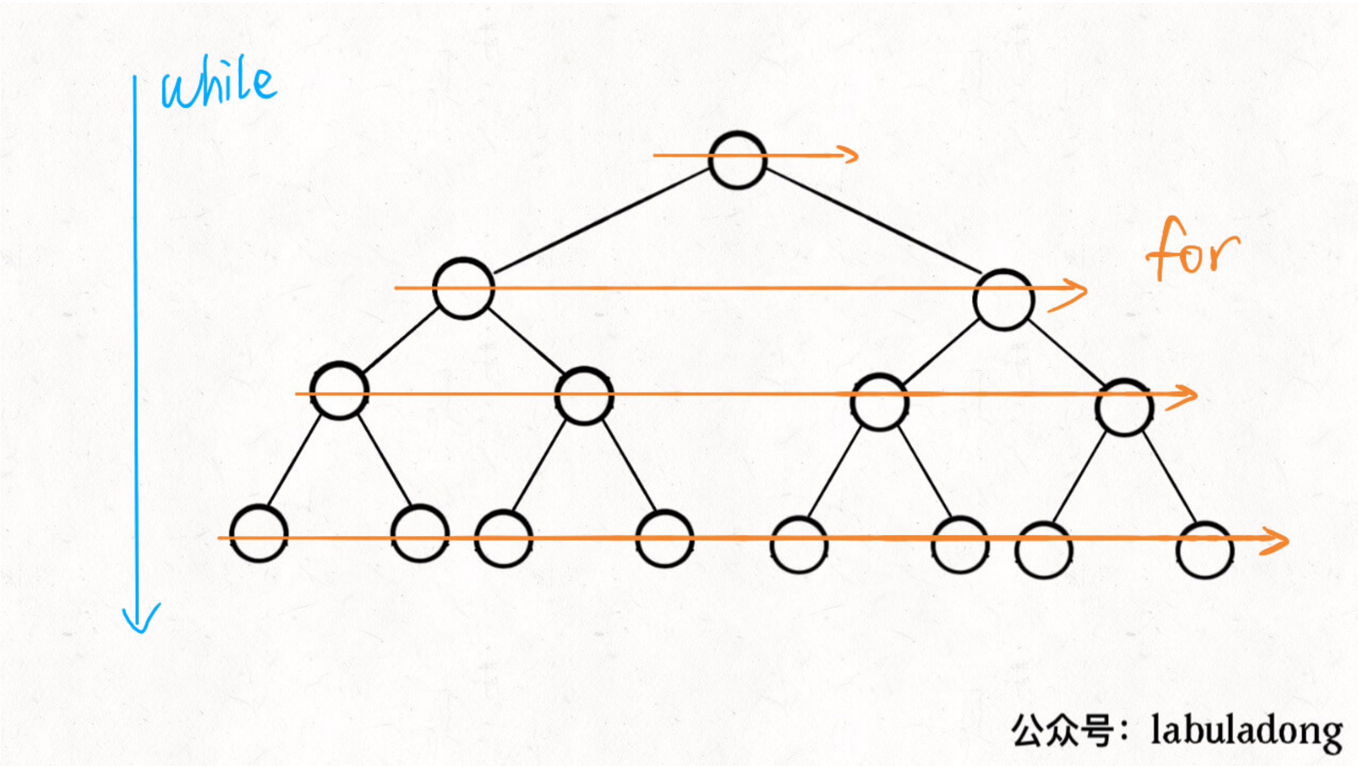
    int depth = 1;
    // 从上到下遍历二叉树的每一层
    while (!q.isEmpty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            printf("节点 %s 在第 %s 层", cur, depth);

            // 将下一层节点放入队列
            if (cur.left != null) {
                q.offer(cur.left);
```

```
            }
            if (cur.right != null) {
                q.offer(cur.right);
            }
        }
        depth++;
    }
}
```

我们先来思考一个问题，注意二叉树的层级遍历 while 循环里面还套了个 for 循环，为什么要这样？

while 循环和 for 循环的配合正是这个遍历框架设计的巧妙之处：



while 循环控制一层一层往下走，for 循环利用 sz 变量控制从左到右遍历每一层二叉树节点。

注意我们代码框架中的 depth 变量，其实就记录了当前遍历到的层数。换句话说，每当我们遍历到一个节点 cur，都知道这个节点属于第几层。

算法题经常会问二叉树的最大深度呀，最小深度呀，层序遍历结果呀，等等问题，所以记录下来这个深度 depth 是有必要的。

基于二叉树的遍历框架，我们又可以扩展出多叉树的层序遍历框架：

```
// 输入一棵多叉树的根节点，层序遍历这棵多叉树
void levelTraverse(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    int depth = 1;
    // 从上到下遍历多叉树的每一层
```

```
    while (!q.isEmpty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            printf("节点 %s 在第 %s 层", cur, depth);

            // 将下一层节点放入队列
            for (TreeNode child : cur.children) {
                q.offer(child);
            }
        }
        depth++;
    }
}
```

基于多叉树的遍历框架，我们又可以扩展出 BFS（广度优先搜索）的算法框架：

```
// 输入起点，进行 BFS 搜索
int BFS(Node start) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路

    q.offer(start); // 将起点加入队列
    visited.add(start);

    int step = 0; // 记录搜索的步数
    while (q not empty) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散一步 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            printf("从 %s 到 %s 的最短距离是 %s", start, cur, step);

            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj()) {
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
            }
        }
        step++;
    }
}
```

如果对 BFS 算法不熟悉，可以看前文 BFS 算法框架，这里只是为了让你做个对比，所谓 BFS 算法，就是把算法问题抽象成一幅「无权图」，然后继续玩二叉树层级遍历那一套罢了。

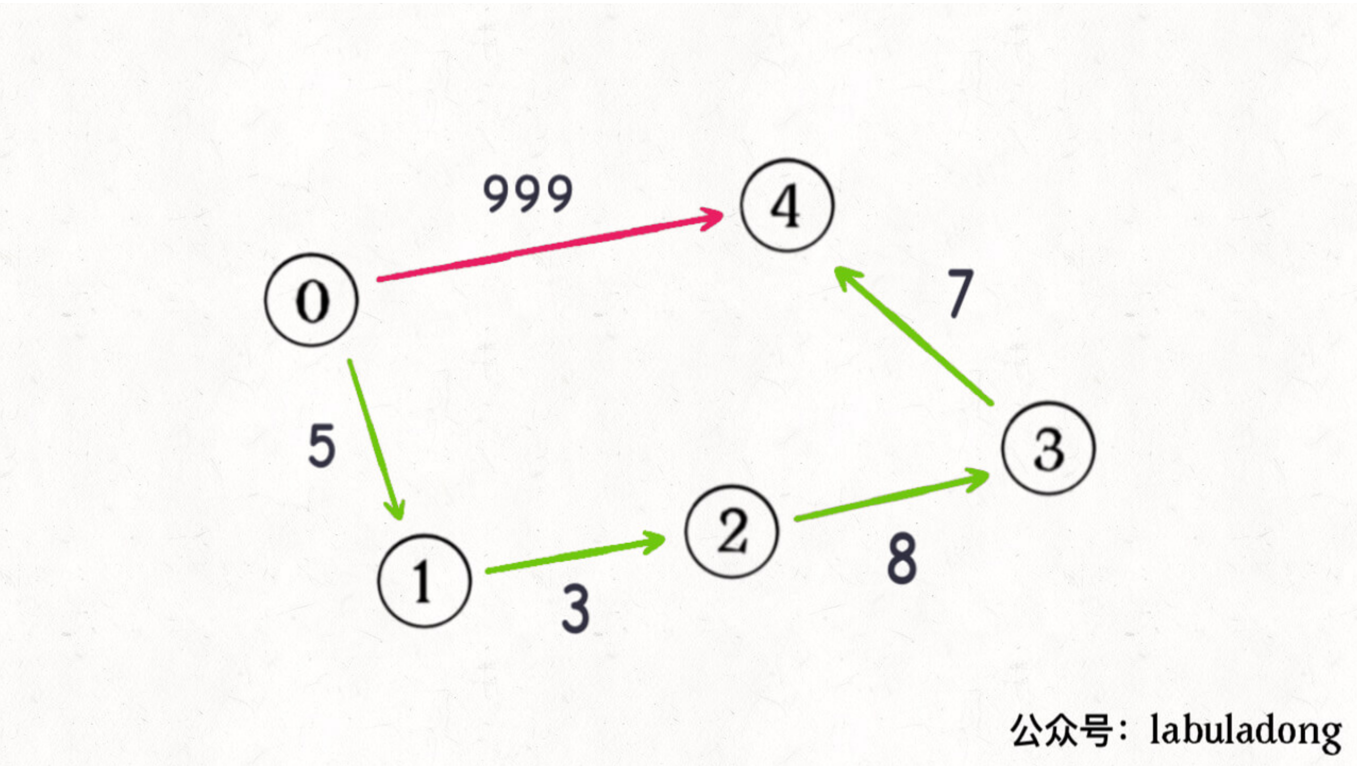
注意，我们的 BFS 算法框架也是 `while` 循环嵌套 `for` 循环的形式，也用了一个 `step` 变量记录 `for` 循环执行的次数，无非就是多用了一个 `visited` 集合记录走过的节点，防止走回头路罢了。

为什么这样呢？

所谓「无权图」，与其说每条「边」没有权重，不如说每条「边」的权重都是 1，从起点 `start` 到任意一个节点之间的路径权重就是它们之间「边」的条数，那可不就是 `step` 变量记录的值么？

再加上 BFS 算法利用 `for` 循环一层一层向外扩散的逻辑和 `visited` 集合防止走回头路的逻辑，当你每次从队列中拿出节点 `cur` 的时候，从 `start` 到 `cur` 的最短权重就是 `step` 记录的步数。

但是，到了「加权图」的场景，事情就没有这么简单了，因为你不能默认每条边的「权重」都是 1 了，这个权重可以是任意正数（Dijkstra 算法要求不能存在负权重边），比如下图的例子：



如果沿用 BFS 算法中的 `step` 变量记录「步数」，显然红色路径一步就可以走到终点，但是这一步的权重很大；正确的最小权重路径应该是绿色的路径，虽然需要走很多步，但是路径权重依然很小。

其实 Dijkstra 和 BFS 算法差不多，不过在讲解 Dijkstra 算法框架之前，我们首先需要对之前的框架进行如下改造：

想办法去掉 `while` 循环里面的 `for` 循环。

为什么？有了刚才的铺垫，这个不难理解，刚才说 `for` 循环是干什么用的来着？

是为了让二叉树一层一层往下遍历，让 BFS 算法一步一步向外扩散，因为这个层数 `depth`，或者这个步数 `step`，在之前的场景中有用。

但现在我们想解决「加权图」中的最短路径问题，「步数」已经没有参考意义了，「路径的权重之和」才有意义，所以这个 `for` 循环可以被去掉。

怎么去掉？就拿二叉树的层级遍历来说，其实你可以直接去掉 `for` 循环相关的代码：

```
// 输入一棵二叉树的根节点，遍历这棵二叉树所有节点
void levelTraverse(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    // 遍历二叉树的每一个节点
    while (!q.isEmpty()) {
        TreeNode cur = q.poll();
        printf("我不知道节点 %s 在第几层", cur);

        // 将子节点放入队列
        if (cur.left != null) {
            q.offer(cur.left);
        }
        if (cur.right != null) {
            q.offer(cur.right);
        }
    }
}
```

但问题是，没有 `for` 循环，你也没办法维护 `depth` 变量了。

如果你想同时维护 `depth` 变量，让每个节点 `cur` 知道自己在第几层，可以想其他办法，比如新建一个 `State` 类，记录每个节点所在的层数：

```
class State {
    // 记录 node 节点的深度
    int depth;
    TreeNode node;

    State(TreeNode node, int depth) {
        this.depth = depth;
        this.node = node;
    }
}

// 输入一棵二叉树的根节点，遍历这棵二叉树所有节点
void levelTraverse(TreeNode root) {
    if (root == null) return 0;
    Queue<State> q = new LinkedList<>();
    q.offer(new State(root, 1));

    // 遍历二叉树的每一个节点
    while (!q.isEmpty()) {
        State cur = q.poll();
        TreeNode cur_node = cur.node;
        int cur_depth = cur.depth;
        printf("节点 %s 在第 %s 层", cur_node, cur_depth);

        // 将子节点放入队列
```

```java
            if (cur_node.left != null) {
                q.offer(new State(cur_node.left, cur_depth + 1));
            }
            if (cur_node.right != null) {
                q.offer(new State(cur_node.right, cur_depth + 1));
            }
        }
    }
```

这样，我们就可以不使用 for 循环也确切地知道每个二叉树节点的深度了。

如果你能够理解上面这段代码，我们就可以来看 Dijkstra 算法的代码框架了。

Dijkstra 算法框架

首先，我们先看一下 Dijkstra 算法的签名：

```java
// 输入一幅图和一个起点 start，计算 start 到其他节点的最短距离
int[] dijkstra(int start, List<Integer>[] graph);
```

输入是一幅图 graph 和一个起点 start，返回是一个记录最短路径权重的数组。

比方说，输入起点 start = 3，函数返回一个 int[] 数组，假设赋值给 distTo 变量，那么从起点 3 到节点 6 的最短路径权重的值就是 distTo[6]。

是的，标准的 Dijkstra 算法会把从起点 start 到所有其他节点的最短路径都算出来。

当然，如果你的需求只是计算从起点 start 到某一个终点 end 的最短路径，那么在标准 Dijkstra 算法上稍作修改就可以更高效地完成这个需求，这个我们后面再说。

其次，我们也需要一个 State 类来辅助算法的运行：

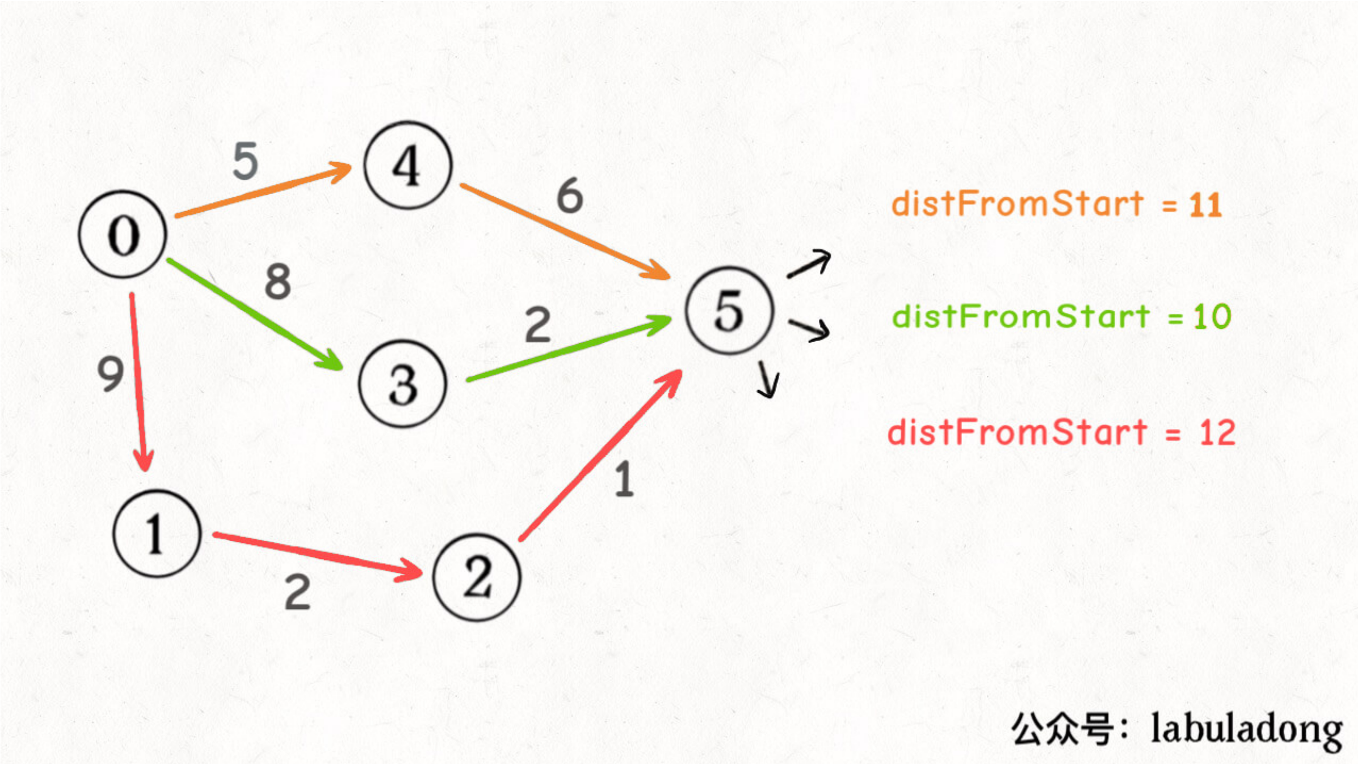
```java
class State {
    // 图节点的 id
    int id;
    // 从 start 节点到当前节点的距离
    int distFromStart;

    State(int id, int distFromStart) {
        this.id = id;
        this.distFromStart = distFromStart;
    }
}
```

类似刚才二叉树的层序遍历，我们也需要用 State 类记录一些额外信息，也就是使用 distFromStart 变量记录从起点 start 到当前这个节点的距离。

刚才说普通 BFS 算法中，根据 BFS 的逻辑和无权图的特点，第一次遇到某个节点所走的步数就是最短距离，所以用一个 `visited` 数组防止走回头路，每个节点只会经过一次。

加权图中的 Dijkstra 算法和无权图中的普通 BFS 算法不同，在 Dijkstra 算法中，你第一次经过某个节点时的路径权重，不见得就是最小的，所以对于同一个节点，我们可能会经过多次，而且每次的 `distFromStart` 可能都不一样，比如下图：



我会经过节点 `5` 三次，每次的 `distFromStart` 值都不一样，那我取 `distFromStart` 最小的那次，不就是从起点 `start` 到节点 `5` 的最短路径权重了么？

好了，明白上面的几点，我们可以来看看 Dijkstra 算法的代码模板。

其实，Dijkstra 可以理解成一个带 dp table（或者说备忘录）的 BFS 算法，伪码如下：

```
// 返回节点 from 到节点 to 之间的边的权重
int weight(int from, int to);

// 输入节点 s 返回 s 的相邻节点
List<Integer> adj(int s);

// 输入一幅图和一个起点 start，计算 start 到其他节点的最短距离
int[] dijkstra(int start, List<Integer>[] graph) {
    // 图中节点的个数
    int V = graph.length;
    // 记录最短路径的权重，你可以理解为 dp table
    // 定义：distTo[i] 的值就是节点 start 到达节点 i 的最短路径权重
    int[] distTo = new int[V];
    // 求最小值，所以 dp table 初始化为正无穷
    Arrays.fill(distTo, Integer.MAX_VALUE);
    // base case，start 到 start 的最短距离就是 0
    distTo[start] = 0;
```

```
    // 优先级队列，distFromStart 较小的排在前面
    Queue<State> pq = new PriorityQueue<>((a, b) -> {
        return a.distFromStart - b.distFromStart;
    });

    // 从起点 start 开始进行 BFS
    pq.offer(new State(start, 0));

    while (!pq.isEmpty()) {
        State curState = pq.poll();
        int curNodeID = curState.id;
        int curDistFromStart = curState.distFromStart;

        if (curDistFromStart > distTo[curNodeID]) {
            // 已经有一条更短的路径到达 curNode 节点了
            continue;
        }
        // 将 curNode 的相邻节点装入队列
        for (int nextNodeID : adj(curNodeID)) {
            // 看看从 curNode 达到 nextNode 的距离是否会更短
            int distToNextNode = distTo[curNodeID] + weight(curNodeID,
nextNodeID);
            if (distTo[nextNodeID] > distToNextNode) {
                // 更新 dp table
                distTo[nextNodeID] = distToNextNode;
                // 将这个节点以及距离放入队列
                pq.offer(new State(nextNodeID, distToNextNode));
            }
        }
    }
    return distTo;
}
```

对比普通的 BFS 算法，你可能会有以下疑问：

1、没有 `visited` 集合记录已访问的节点，所以一个节点会被访问多次，会被多次加入队列，那会不会导致队列永远不为空，造成死循环？

2、为什么用优先级队列 `PriorityQueue` 而不是 `LinkedList` 实现的普通队列？为什么要按照 `distFromStart` 的值来排序？

3、如果我只想计算起点 `start` 到某一个终点 `end` 的最短路径，是否可以修改算法，提升一些效率？

我们先回答第一个问题，为什么这个算法不用 `visited` 集合也不会死循环。

对于这类问题，我教你一个思考方法：

循环结束的条件是队列为空，那么你就要注意看什么时候往队列里放元素（调用 `offer`）方法，再注意看什么时候从队列往外拿元素（调用 `poll` 方法）。

`while` 循环每执行一次，都会往外拿一个元素，但想往队列里放元素，可就有很多限制了，必须满足下面这个条件：

```java
// 看看从 curNode 达到 nextNode 的距离是否会更短
if (distTo[nextNodeID] > distToNextNode) {
    // 更新 dp table
    distTo[nextNodeID] = distToNextNode;
    pq.offer(new State(nextNodeID, distToNextNode));
}
```

这也是为什么我说 `distTo` 数组可以理解成我们熟悉的 dp table，因为这个算法逻辑就是在不断的最小化 `distTo` 数组中的元素：

如果你能让到达 `nextNodeID` 的距离更短，那就更新 `distTo[nextNodeID]` 的值，让你入队，否则的话对不起，不让入队。

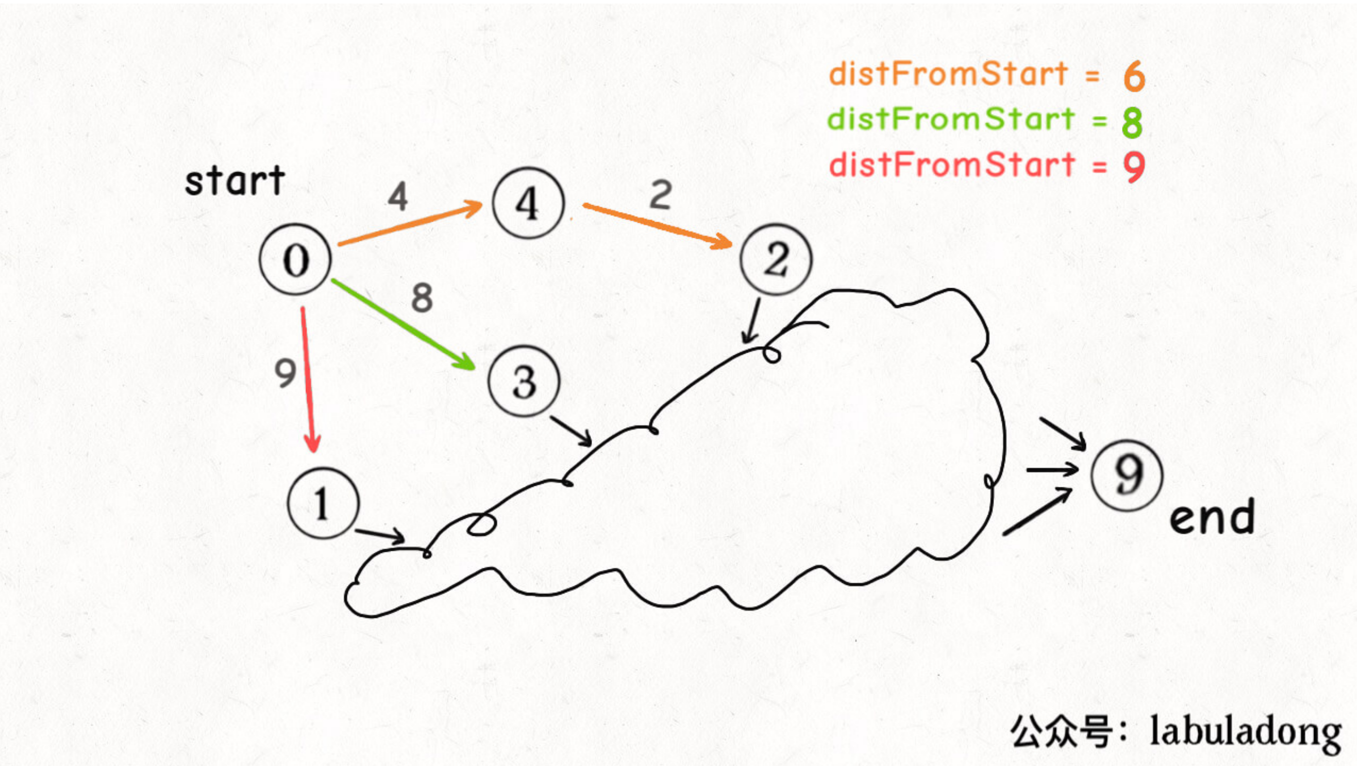
因为两个节点之间的最短距离（路径权重）肯定是一个确定的值，不可能无限减小下去，所以队列一定会空，队列空了之后，`distTo` 数组中记录的就是从 `start` 到其他节点的最短距离。

接下来解答第二个问题，为什么要用 `PriorityQueue` 而不是 `LinkedList` 实现的普通队列？

如果你非要用普通队列，其实也没问题的，你可以直接把 `PriorityQueue` 改成 `LinkedList`，也能得到正确答案，但是效率会低很多。

Dijkstra 算法使用优先级队列，主要是为了效率上的优化，类似一种贪心算法的思路。

为什么说是一种贪心思路呢，比如说下面这种情况，你想计算从起点 `start` 到终点 `end` 的最短路径权重：



假设你当前只遍历了图中的这几个节点，那么你下一步准备遍历那个节点？这三条路径都可能成为最短路径的一部分，**但你觉得哪条路径更有「潜力」成为最短路径中的一部分**？

从目前的情况来看，显然橙色路径的可能性更大嘛，所以我们希望节点 `2` 排在队列靠前的位置，优先被拿出来向后遍历。

所以我们使用 PriorityQueue 作为队列，让 distFromStart 的值较小的节点排在前面，这就类似我们之前讲 贪心算法 说到的贪心思路，可以很大程度上优化算法的效率。

大家应该听过 Bellman-Ford 算法，这个算法是一种更通用的最短路径算法，因为它可以处理带有负权重边的图，Bellman-Ford 算法逻辑和 Dijkstra 算法非常类似，用到的就是普通队列，本文就提一句，后面有空再具体写。

接下来说第三个问题，如果只关心起点 start 到某一个终点 end 的最短路径，是否可以修改代码提升算法效率。

肯定可以的，因为我们标准 Dijkstra 算法会算出 start 到所有其他节点的最短路径，你只想计算到 end 的最短路径，相当于减少计算量，当然可以提升效率。

需要在代码中做的修改也非常少，只要改改函数签名，再加个 if 判断就行了：

```java
// 输入起点 start 和终点 end，计算起点到终点的最短距离
int dijkstra(int start, int end, List<Integer>[] graph) {

    // ...

    while (!pq.isEmpty()) {
        State curState = pq.poll();
        int curNodeID = curState.id;
        int curDistFromStart = curState.distFromStart;

        // 在这里加一个判断就行了，其他代码不用改
        if (curNodeID == end) {
            return curDistFromStart;
        }

        if (curDistFromStart > distTo[curNodeID]) {
            continue;
        }

        // ...
    }

    // 如果运行到这里，说明从 start 无法走到 end
    return Integer.MAX_VALUE;
}
```

因为优先级队列自动排序的性质，**每次**从队列里面拿出来的都是 distFromStart 值最小的，所以当你**第一次**从队列中拿出终点 end 时，此时的 distFromStart 对应的值就是从 start 到 end 的最短距离。

这个算法较之前的实现提前 return 了，所以效率有一定的提高。

时间复杂度分析

Dijkstra 算法的时间复杂度是多少？你去网上查，可能会告诉你是 O(ElogV)，其中 E 代表图中边的条数，V 代表图中节点的个数。

因为理想情况下优先级队列中最多装 `V` 个节点，对优先级队列的操作次数和 `E` 成正比，所以整体的时间复杂度就是 `O(ElogV)`。

不过这是理想情况，Dijkstra 算法的代码实现有很多版本，不同编程语言或者不同数据结构 API 都会导致算法的时间复杂度发生一些改变。

比如本文实现的 Dijkstra 算法，使用了 Java 的 `PriorityQueue` 这个数据结构，这个容器类底层使用二叉堆实现，但没有提供通过索引操作队列中元素的 API，所以队列中会有重复的节点，最多可能有 `E` 个节点存在队列中。

所以本文实现的 Dijkstra 算法复杂度并不是理想情况下的 `O(ElogV)`，而是 `O(ElogE)`，可能会略大一些，因为图中边的条数一般是大于节点的个数的。

不过就对数函数来说，就算真数大一些，对数函数的结果也大不了多少，所以这个算法实现的实际运行效率也是很高的，以上只是理论层面的时间复杂度分析，供大家参考。

秒杀三道题目

以上说了 Dijkstra 算法的框架，下面我们套用这个框架做几道题，实践出真知。

第一题是力扣第 743 题「网络延迟时间」，题目如下：

743. 网络延迟时间 labuladong 题解 思路

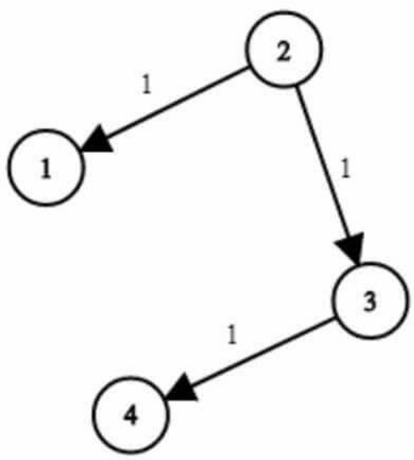
难度 中等 425 收藏 分享 切换为英文 接收动态 反馈

有 `n` 个网络节点，标记为 `1` 到 `n`。

给你一个列表 `times`，表示信号经过 **有向** 边的传递时间。 `times[i] = (ui, vi, wi)`，其中 `ui` 是源节点，`vi` 是目标节点， `wi` 是一个信号从源节点传递到目标节点的时间。

现在，从某个节点 `K` 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点收到信号，返回 `-1` 。

示例 1：




```
输入：times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
输出：2
```

函数签名如下：

```
// times 记录边和权重，n 为节点个数（从 1 开始），k 为起点
// 计算从 k 发出的信号至少需要多久传遍整幅图
int networkDelayTime(int[][] times, int n, int k)
```

让你求所有节点都收到信号的时间，你把所谓的传递时间看做距离，实际上就是问你「从节点 `k` 到其他所有节点的最短路径中，最长的那条最短路径距离是多少」，说白了就是让你算从节点 `k` 出发到其他所有节点的最短路径，就是标准的 Dijkstra 算法。

在用 Dijkstra 之前，别忘了要满足一些条件，加权有向图，没有负权重边，OK，可以用 Dijkstra 算法计算最短路径。

根据我们之前 Dijkstra 算法的框架，我们可以写出下面代码：

```
int networkDelayTime(int[][] times, int n, int k) {
    // 节点编号是从 1 开始的，所以要一个大小为 n + 1 的邻接表
```

```java
    List<int[]>[] graph = new LinkedList[n + 1];
    for (int i = 1; i <= n; i++) {
        graph[i] = new LinkedList<>();
    }
    // 构造图
    for (int[] edge : times) {
        int from = edge[0];
        int to = edge[1];
        int weight = edge[2];
        // from -> List<(to, weight)>
        // 邻接表存储图结构，同时存储权重信息
        graph[from].add(new int[]{to, weight});
    }
    // 启动 dijkstra 算法计算以节点 k 为起点到其他节点的最短路径
    int[] distTo = dijkstra(k, graph);

    // 找到最长的那一条最短路径
    int res = 0;
    for (int i = 1; i < distTo.length; i++) {
        if (distTo[i] == Integer.MAX_VALUE) {
            // 有节点不可达，返回 -1
            return -1;
        }
        res = Math.max(res, distTo[i]);
    }
    return res;
}

// 输入一个起点 start，计算从 start 到其他节点的最短距离
int[] dijkstra(int start, List<int[]>[] graph) {}
```

上述代码首先利用题目输入的数据转化成邻接表表示一幅图，接下来我们可以直接套用 Dijkstra 算法的框架：

```java
class State {
    // 图节点的 id
    int id;
    // 从 start 节点到当前节点的距离
    int distFromStart;

    State(int id, int distFromStart) {
        this.id = id;
        this.distFromStart = distFromStart;
    }
}

// 输入一个起点 start，计算从 start 到其他节点的最短距离
int[] dijkstra(int start, List<int[]>[] graph) {
    // 定义：distTo[i] 的值就是起点 start 到达节点 i 的最短路径权重
    int[] distTo = new int[graph.length];
    Arrays.fill(distTo, Integer.MAX_VALUE);
```

```java
    // base case, start 到 start 的最短距离就是 0
    distTo[start] = 0;

    // 优先级队列，distFromStart 较小的排在前面
    Queue<State> pq = new PriorityQueue<>((a, b) -> {
        return a.distFromStart - b.distFromStart;
    });
    // 从起点 start 开始进行 BFS
    pq.offer(new State(start, 0));

    while (!pq.isEmpty()) {
        State curState = pq.poll();
        int curNodeID = curState.id;
        int curDistFromStart = curState.distFromStart;

        if (curDistFromStart > distTo[curNodeID]) {
            continue;
        }

        // 将 curNode 的相邻节点装入队列
        for (int[] neighbor : graph[curNodeID]) {
            int nextNodeID = neighbor[0];
            int distToNextNode = distTo[curNodeID] + neighbor[1];
            // 更新 dp table
            if (distTo[nextNodeID] > distToNextNode) {
                distTo[nextNodeID] = distToNextNode;
                pq.offer(new State(nextNodeID, distToNextNode));
            }
        }
    }
    return distTo;
}
```

你对比之前说的代码框架，只要稍稍修改，就可以把这道题目解决了。

感觉这道题完全没有难度，下面我们再看一道题目，力扣第 1631 题「最小体力消耗路径」：

1631. 最小体力消耗路径 labuladong 题解 思路

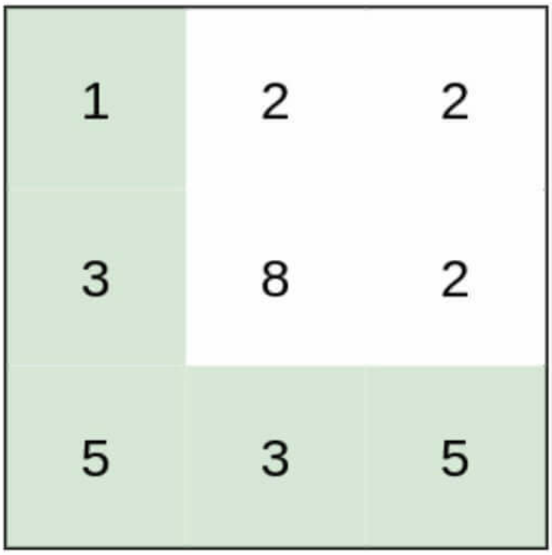
难度 中等 233 收藏 分享 切换为英文 接收动态 反馈

你准备参加一场远足活动。给你一个二维 `rows x columns` 的地图 `heights` ，其中 `heights[row][col]` 表示格子 `(row, col)` 的高度。一开始你在最左上角的格子 `(0, 0)` ，且你希望去最右下角的格子 `(rows-1, columns-1)` （注意下标从 **0** 开始编号）。你每次可以往 **上**，**下**，**左**，**右** 四个方向之一移动，你想要找到耗费 **体力** 最小的一条路径。

一条路径耗费的 **体力值** 是路径上相邻格子之间 **高度差绝对值** 的 **最大值** 决定的。

请你返回从左上角走到右下角的最小 **体力消耗值** 。

示例 1：




```
输入：heights = [[1,2,2],[3,8,2],[5,3,5]]
输出：2
解释：路径 [1,3,5,3,5] 连续格子的差值绝对值最大为 2 。
这条路径比路径 [1,2,2,2,5] 更优，因为另一条路径差值最大值为 3 。
```

函数签名如下：

```
// 输入一个二维矩阵，计算从左上角到右下角的最小体力消耗
int minimumEffortPath(int[][] heights);
```

我们常见的二维矩阵题目，如果让你从左上角走到右下角，比较简单的题一般都会限制你只能向右或向下走，但这道题可没有限制哦，你可以上下左右随便走，只要路径的「体力消耗」最小就行。

如果你把二维数组中每个 `(x, y)` 坐标看做一个节点，它的上下左右坐标就是相邻节点，它对应的值和相邻坐标对应的值之差的绝对值就是题目说的「体力消耗」，你就可以理解为边的权重。

这样一想，是不是就在让你以左上角坐标为起点，以右下角坐标为终点，计算起点到终点的最短路径？Dijkstra 算法是不是可以做到？

```java
// 输入起点 start 和终点 end，计算起点到终点的最短距离
int dijkstra(int start, int end, List<Integer>[] graph)
```

只不过，这道题中评判一条路径是长还是短的标准不再是路径经过的权重总和，而是路径经过的权重最大值。

明白这一点，再想一下使用 Dijkstra 算法的前提，加权有向图，没有负权重边，求最短路径，OK，可以使用，咱们来套框架。

二维矩阵抽象成图，我们先实现一下图的 `adj` 方法，之后的主要逻辑会清晰一些：

```java
// 方向数组，上下左右的坐标偏移量
int[][] dirs = new int[][]{{0,1}, {1,0}, {0,-1}, {-1,0}};

// 返回坐标 (x, y) 的上下左右相邻坐标
List<int[]> adj(int[][] matrix, int x, int y) {
    int m = matrix.length, n = matrix[0].length;
    // 存储相邻节点
    List<int[]> neighbors = new ArrayList<>();
    for (int[] dir : dirs) {
        int nx = x + dir[0];
        int ny = y + dir[1];
        if (nx >= m || nx < 0 || ny >= n || ny < 0) {
            // 索引越界
            continue;
        }
        neighbors.add(new int[]{nx, ny});
    }
    return neighbors;
}
```

类似的，我们现在认为一个二维坐标 `(x, y)` 是图中的一个节点，所以这个 `State` 类也需要修改一下：

```java
class State {
    // 矩阵中的一个位置
    int x, y;
    // 从起点 (0, 0) 到当前位置的最小体力消耗（距离）
    int effortFromStart;

    State(int x, int y, int effortFromStart) {
        this.x = x;
        this.y = y;
        this.effortFromStart = effortFromStart;
    }
}
```

接下来，就可以套用 Dijkstra 算法的代码模板了：

```java
// Dijkstra 算法，计算 (0, 0) 到 (m - 1, n - 1) 的最小体力消耗
int minimumEffortPath(int[][] heights) {
    int m = heights.length, n = heights[0].length;
    // 定义：从 (0, 0) 到 (i, j) 的最小体力消耗是 effortTo[i][j]
    int[][] effortTo = new int[m][n];
    // dp table 初始化为正无穷
    for (int i = 0; i < m; i++) {
        Arrays.fill(effortTo[i], Integer.MAX_VALUE);
    }
    // base case，起点到起点的最小消耗就是 0
    effortTo[0][0] = 0;

    // 优先级队列，effortFromStart 较小的排在前面
    Queue<State> pq = new PriorityQueue<>((a, b) -> {
        return a.effortFromStart - b.effortFromStart;
    });

    // 从起点 (0, 0) 开始进行 BFS
    pq.offer(new State(0, 0, 0));

    while (!pq.isEmpty()) {
        State curState = pq.poll();
        int curX = curState.x;
        int curY = curState.y;
        int curEffortFromStart = curState.effortFromStart;

        // 到达终点提前结束
        if (curX == m - 1 && curY == n - 1) {
            return curEffortFromStart;
        }

        if (curEffortFromStart > effortTo[curX][curY]) {
            continue;
        }
        // 将 (curX, curY) 的相邻坐标装入队列
        for (int[] neighbor : adj(heights, curX, curY)) {
            int nextX = neighbor[0];
            int nextY = neighbor[1];
            // 计算从 (curX, curY) 达到 (nextX, nextY) 的消耗
            int effortToNextNode = Math.max(
                effortTo[curX][curY],
                Math.abs(heights[curX][curY] - heights[nextX][nextY])
            );
            // 更新 dp table
            if (effortTo[nextX][nextY] > effortToNextNode) {
                effortTo[nextX][nextY] = effortToNextNode;
                pq.offer(new State(nextX, nextY, effortToNextNode));
            }
        }
    }
    // 正常情况不会达到这个 return
```

```
        return -1;
    }
```

你看，稍微改一改代码模板，这道题就解决了。

最后看一道题吧，力扣第 1514 题「概率最大的路径」，看下题目：

1514. 概率最大的路径 labuladong 题解 思路

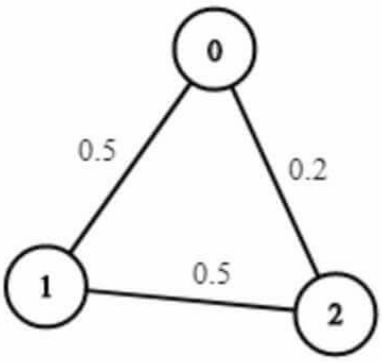
难度 中等 68 收藏 分享 切换为英文 接收动态 反馈

给你一个由 `n` 个节点（下标从 0 开始）组成的无向加权图，该图由一个描述边的列表组成，其中 `edges[i] = [a, b]` 表示连接节点 a 和 b 的一条无向边，且该边遍历成功的概率为 `succProb[i]` 。

指定两个节点分别作为起点 `start` 和终点 `end` ，请你找出从起点到终点成功概率最大的路径，并返回其成功概率。

如果不存在从 `start` 到 `end` 的路径，请 **返回 0** 。只要答案与标准答案的误差不超过 **1e-5** ，就会被视作正确答案。

示例 1：




```
输入：n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start =
0, end = 2
输出：0.25000
解释：从起点到终点有两条路径，其中一条的成功概率为 0.2 ，而另一条为 0.5 * 0.5 = 0.25
```

函数签名如下：

```
// 输入一幅无向图，边上的权重代表概率，返回从 start 到达 end 最大的概率
double maxProbability(int n, int[][] edges, double[] succProb, int start,
int end)
```

我说这题一看就是 Dijkstra 算法，但聪明的你肯定会反驳我：

1、这题给的是无向图，也可以用 Dijkstra 算法吗？

2、更重要的是，Dijkstra 算法计算的是最短路径，计算的是最小值，这题让你计算最大概率是一个最大值，怎么可能用 Dijkstra 算法呢？

问得好！

首先关于有向图和无向图，前文 图算法基础 说过，无向图本质上可以认为是「双向图」，从而转化成有向图。

重点说说最大值和最小值这个问题，其实 Dijkstra 和很多最优化算法一样，计算的是「最优值」，这个最优值可能是最大值，也可能是最小值。

标准 Dijkstra 算法是计算最短路径的，但你有想过为什么 Dijkstra 算法不允许存在负权重边么？

因为 Dijkstra 计算最短路径的正确性依赖一个前提：路径中每增加一条边，路径的总权重就会增加。

这个前提的数学证明大家有兴趣可以自己搜索一下，我这里只说结论，其实你把这个结论反过来也是 OK 的：

如果你想计算最长路径，路径中每增加一条边，路径的总权重就会减少，要是能够满足这个条件，也可以用 Dijkstra 算法。

你看这道题是不是符合这个条件？边和边之间是乘法关系，每条边的概率都是小于 1 的，所以肯定会越乘越小。

只不过，这道题的解法要把优先级队列的排序顺序反过来，一些 if 大小判断也要反过来，我们直接看解法代码吧：

```
double maxProbability(int n, int[][] edges, double[] succProb, int start,
int end) {
    List<double[]>[] graph = new LinkedList[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new LinkedList<>();
    }
    // 构造邻接表结构表示图
    for (int i = 0; i < edges.length; i++) {
        int from = edges[i][0];
        int to = edges[i][1];
        double weight = succProb[i];
        // 无向图就是双向图；先把 int 统一转成 double，待会再转回来
        graph[from].add(new double[]{(double)to, weight});
        graph[to].add(new double[]{(double)from, weight});
    }

    return dijkstra(start, end, graph);
}

class State {
    // 图节点的 id
    int id;
    // 从 start 节点到达当前节点的概率
    double probFromStart;
```

```java
    State(int id, double probFromStart) {
        this.id = id;
        this.probFromStart = probFromStart;
    }
}

double dijkstra(int start, int end, List<double[]>[] graph) {
    // 定义：probTo[i] 的值就是节点 start 到达节点 i 的最大概率
    double[] probTo = new double[graph.length];
    // dp table 初始化为一个取不到的最小值
    Arrays.fill(probTo, -1);
    // base case，start 到 start 的概率就是 1
    probTo[start] = 1;

    // 优先级队列，probFromStart 较大的排在前面
    Queue<State> pq = new PriorityQueue<>((a, b) -> {
        return Double.compare(b.probFromStart, a.probFromStart);
    });
    // 从起点 start 开始进行 BFS
    pq.offer(new State(start, 1));

    while (!pq.isEmpty()) {
        State curState = pq.poll();
        int curNodeID = curState.id;
        double curProbFromStart = curState.probFromStart;

        // 遇到终点提前返回
        if (curNodeID == end) {
            return curProbFromStart;
        }

        if (curProbFromStart < probTo[curNodeID]) {
            // 已经有一条概率更大的路径到达 curNode 节点了
            continue;
        }
        // 将 curNode 的相邻节点装入队列
        for (double[] neighbor : graph[curNodeID]) {
            int nextNodeID = (int)neighbor[0];
            // 看看从 curNode 达到 nextNode 的概率是否会更大
            double probToNextNode = probTo[curNodeID] * neighbor[1];
            if (probTo[nextNodeID] < probToNextNode) {
                probTo[nextNodeID] = probToNextNode;
                pq.offer(new State(nextNodeID, probToNextNode));
            }
        }
    }
    // 如果到达这里，说明从 start 开始无法到达 end，返回 0
    return 0.0;
}
```

好了，到这里本文就结束了，总共 6000 多字，这三道例题都是比较困难的，如果你能够看到这里，真得给你鼓掌。

其实前文 毕业旅行省钱算法 中讲过限制之下的最小路径问题，当时是使用动态规划思路解决的，但文末也给了 Dijkstra 算法代码，仅仅在本文模板的基础上做了一些变换，你理解本文后可以对照着去看看那道题目。

最后还是那句话，做题在质不在量，希望大家能够透彻理解最基本的数据结构，以不变应万变。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



众里寻他千百度：名流问题

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

277. 搜索名人（中等）

今天来讨论经典的「名流问题」：

给你 `n` 个人的社交关系（你知道任意两个人之间是否认识），然后请你找出这些人中的「名人」。

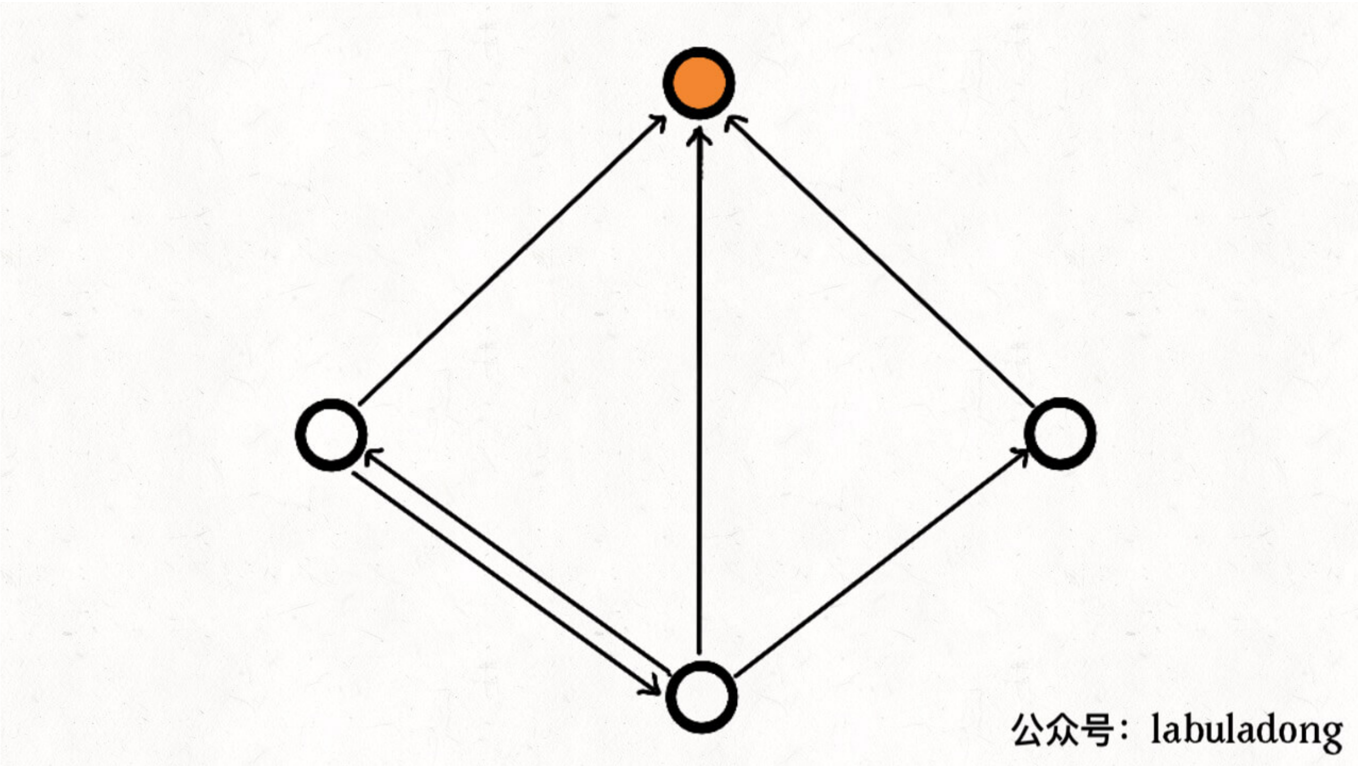
所谓「名人」有两个条件：

1、所有其他人都认识「名人」。

2、「名人」不认识任何其他人。

这是一个图相关的算法问题，社交关系嘛，本质上就可以抽象成一幅图。

如果把每个人看做图中的节点，「认识」这种关系看做是节点之间的有向边，那么名人就是这幅图中一个特殊的节点：



这个节点没有一条指向其他节点的有向边；且其他所有节点都有一条指向这个节点的有向边。

或者说的专业一点，名人节点的出度为 0，入度为 `n - 1`。

那么，这 `n` 个人的社交关系是如何表示的呢？

前文 图论算法基础 说过，图有两种存储形式，一种是邻接表，一种是邻接矩阵，邻接表的主要优势是节约存储空间；邻接矩阵的主要优势是可以迅速判断两个节点是否相邻。

对于名人问题，显然会经常需要判断两个人之间是否认识，也就是两个节点是否相邻，所以我们可以用邻接表来表示人和人之间的社交关系。

那么，把名流问题描述成算法的形式就是这样的：

给你输入一个大小为 `n x n` 的二维数组（邻接矩阵） `graph` 表示一幅有 `n` 个节点的图，每个人都是图中的一个节点，编号为 `0` 到 `n - 1`。

如果 `graph[i][j] == 1` 代表第 `i` 个人认识第 `j` 个人，如果 `graph[i][j] == 0` 代表第 `i` 个人不认识第 `j` 个人。

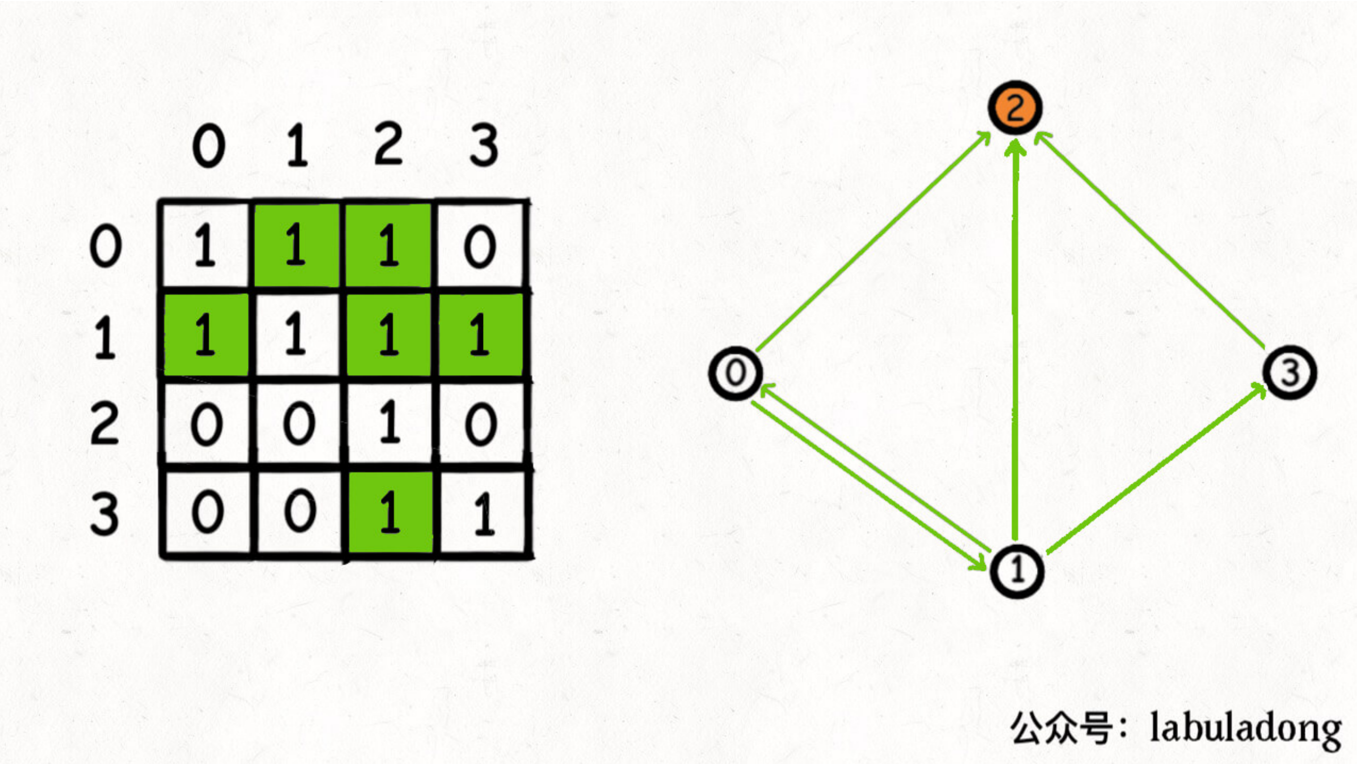
有了这幅图表示人与人之间的关系，请你计算，这 `n` 个人中，是否存在「名人」？

如果存在，算法返回这个名人的编号，如果不存在，算法返回 -1。

函数签名如下：

```
int findCelebrity(int[][] graph);
```

比如输入的邻接矩阵长这样：



那么算法应该返回 2。

力扣第 277 题「搜寻名人」就是这个经典问题，不过并不是直接把邻接矩阵传给你，而是只告诉你总人数 `n`，同时提供一个 API `knows` 来查询人和人之间的社交关系：

```
// 可以直接调用，能够返回 i 是否认识 j
boolean knows(int i, int j);

// 请你实现：返回「名人」的编号
int findCelebrity(int n) {
    // todo
}
```

很明显，`knows` API 本质上还是在访问邻接矩阵。为了简单起见，我们后面就按力扣的题目形式来探讨一下这个经典问题。

暴力解法

我们拍拍脑袋就能写出一个简单粗暴的算法：

```
int findCelebrity(int n) {
    for (int cand = 0; cand < n; cand++) {
        int other;
        for (other = 0; other < n; other++) {
            if (cand == other) continue;
            // 保证其他人都认识 cand，且 cand 不认识任何其他人
            // 否则 cand 就不可能是名人
            if (knows(cand, other) || !knows(other, cand)) {
                break;
```

```
                }
            }
            if (other == n) {
                // 找到名人
                return cand;
            }
        }
        // 没有一个人符合名人特性
        return -1;
    }
```

cand 是候选人（candidate）的缩写，我们的暴力算法就是从头开始穷举，把每个人都视为候选人，判断是否符合「名人」的条件。

刚才也说了，knows 函数底层就是在访问一个二维的邻接矩阵，一次调用的时间复杂度是 O(1)，所以这个暴力解法整体的最坏时间复杂度是 O(N^2)。

那么，是否有其他高明的办法来优化时间复杂度呢？其实是有优化空间的，你想想，我们现在最耗时的地方在哪里？

对于每一个候选人 cand，我们都要用一个内层 for 循环去判断这个 cand 到底符不符合「名人」的条件。

这个内层 for 循环看起来就蠢，虽然判断一个人「是名人」必须用一个 for 循环，但判断一个人「不是名人」就不用这么麻烦了。

因为「名人」的定义保证了「名人」的唯一性，所以我们可以利用排除法，先排除那些显然不是「名人」的人，从而避免 for 循环的嵌套，降低时间复杂度。

优化解法

我再重复一遍所谓「名人」的定义：

1、所有其他人都认识名人。

2、名人不认识任何其他人。

这个定义就很有意思，它保证了人群中最多有一个名人。

这很好理解，如果有两个人同时是名人，那么这两条定义就自相矛盾了。

换句话说，只要观察任意两个候选人的关系，我一定能确定其中的一个人不是名人，把他排除。

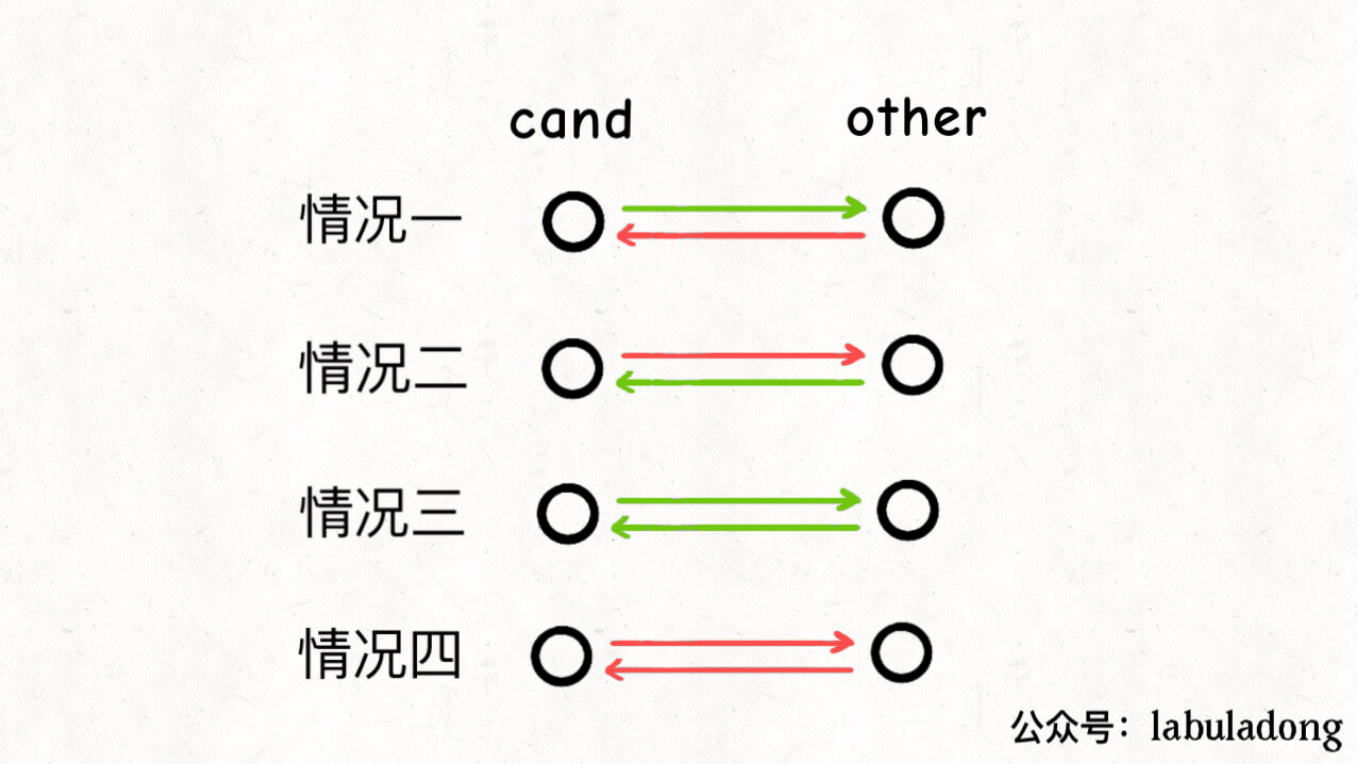
至于另一个候选人是不是名人，只看两个人的关系肯定是不能确定的，但这不重要，重要的是排除掉一个必然不是名人的候选人，缩小了包围圈。

这是优化的核心，也是比较难理解的，所以我们先来说说为什么观察任意两个候选人的关系，就能排除掉一个。

你想想，两个人之间的关系可能是什么样的？

无非就是四种：你认识我我不认识你，我认识你你不认识我，咱俩互相认识，咱两互相不认识。

如果把人比作节点，红色的有向边表示不认识，绿色的有向边表示认识，那么两个人的关系无非是如下四种情况：



不妨认为这两个人的编号分别是 `cand` 和 `other`，然后我们逐一分析每种情况，看看怎么排除掉一个人。

对于情况一，`cand` 认识 `other`，所以 `cand` 肯定不是名人，排除。因为名人不可能认识别人。

对于情况二，`other` 认识 `cand`，所以 `other` 肯定不是名人，排除。

对于情况三，他俩互相认识，肯定都不是名人，可以随便排除一个。

对于情况四，他俩互不认识，肯定都不是名人，可以随便排除一个。因为名人应该被所有其他人认识。

综上，只要观察任意两个之间的关系，就至少能确定一个人不是名人，上述情况判断可以用如下代码表示：

```
if (knows(cand, other) || !knows(other, cand)) {
    // cand 不可能是名人
} else {
    // other 不可能是名人
}
```

如果能够理解这一个特点，那么写出优化解法就简单了。

我们可以不断从候选人中选两个出来，然后排除掉一个，直到最后只剩下一个候选人，这时候再使用一个 for 循环判断这个候选人是否是货真价实的「名人」。

这个思路的完整代码如下：

```
int findCelebrity(int n) {
    if (n == 1) return 0;
```

```java
    // 将所有候选人装进队列
    LinkedList<Integer> q = new LinkedList<>();
    for (int i = 0; i < n; i++) {
        q.addLast(i);
    }
    // 一直排除，直到只剩下一个候选人停止循环
    while (q.size() >= 2) {
        // 每次取出两个候选人，排除一个
        int cand = q.removeFirst();
        int other = q.removeFirst();
        if (knows(cand, other) || !knows(other, cand)) {
            // cand 不可能是名人，排除，让 other 归队
            q.addFirst(other);
        } else {
            // other 不可能是名人，排除，让 cand 归队
            q.addFirst(cand);
        }
    }

    // 现在排除得只剩一个候选人，判断他是否真的是名人
    int cand = q.removeFirst();
    for (int other = 0; other < n; other++) {
        if (other == cand) {
            continue;
        }
        // 保证其他人都认识 cand，且 cand 不认识任何其他人
        if (!knows(other, cand) || knows(cand, other)) {
            return -1;
        }
    }
    // cand 是名人
    return cand;
}
```

这个算法避免了嵌套 for 循环，时间复杂度降为 O(N) 了，不过引入了一个队列来存储候选人集合，使用了 O(N) 的空间复杂度。

> PS：`LinkedList` 的作用只是充当一个容器把候选人装起来，每次找出两个进行比较和淘汰，但至于具体找出哪两个，都是无所谓的，也就是说候选人归队的顺序无所谓，我们用的是 `addFirst` 只是方便后续的优化，你完全可以用 `addLast`，结果都是一样的。

是否可以进一步优化，把空间复杂度也优化掉?

最终解法

如果你能够理解上面的优化解法，其实可以不需要额外的空间解决这个问题，代码如下：

```java
int findCelebrity(int n) {
    // 先假设 cand 是名人
    int cand = 0;
    for (int other = 1; other < n; other++) {
```

```
            if (!knows(other, cand) || knows(cand, other)) {
                // cand 不可能是名人，排除
                // 假设 other 是名人
                cand = other;
            } else {
                // other 不可能是名人，排除
                // 什么都不用做，继续假设 cand 是名人
            }
        }

        // 现在的 cand 是排除的最后结果，但不能保证一定是名人
        for (int other = 0; other < n; other++) {
            if (cand == other) continue;
            // 需要保证其他人都认识 cand，且 cand 不认识任何其他人
            if (!knows(other, cand) || knows(cand, other)) {
                return -1;
            }
        }

        return cand;
    }
```

我们之前的解法用到了 `LinkedList` 充当一个队列，用于存储候选人集合，而这个优化解法利用 `other` 和 `cand` 的交替变化，模拟了我们之前操作队列的过程，避免了使用额外的存储空间。

现在，解决名人问题的解法时间复杂度为 O(N)，空间复杂度为 O(1)，已经是最优解法了。

如果你有高质量的问题，可以 [点击这里](#) 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



霸剑篇、暴力搜索算法



计算机除了穷举之外啥也不会，所谓算法就是考察你会不会穷举，能不能聪明地穷举。

宽泛地讲，for 循环遍历一遍数组，这也叫穷举，但本章说的暴力穷举主要包括深度优先（DFS）算法和广度优先（BFS）算法。

其中，DFS 算法和回溯算法可以说是师出同门，大同小异，图论算法基础 探讨过这个问题，区别仅仅在于根节点是否被遍历到而已。

且 BFS 算法常见于求最值的场景，因为 BFS 的算法逻辑保证了算法第一次到达目标时的代价是最小的。

公众号标签：暴力搜索算法

3.1 DFS 算法/回溯算法

回溯算法的效率一般不高，但却是最好用的算法。

因为回溯算法就是典型的暴力穷举算法嘛，简单粗暴，如果你笔试的时候不会做一道题，那就尝试用回溯算法硬上，超时没关系，多少能捞点分回来。

回溯算法框架如下：

```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

回溯算法解题套路框架

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong



学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

46. 全排列（中等）

51. N 皇后（困难）

本文有视频版：回溯算法框架套路详解

这篇文章是很久之前的一篇 回溯算法详解 的进阶版，之前那篇不够清楚，就不必看了，看这篇就行。把框架给你讲清楚，你会发现回溯算法问题都是一个套路。

本文解决几个问题：

回溯算法是什么？解决回溯算法相关的问题有什么技巧？如何学习回溯算法？回溯算法代码是否有规律可循？

其实回溯算法其实就是我们常说的 DFS 算法，本质上就是一种暴力穷举算法。

废话不多说，直接上回溯算法框架。**解决一个回溯问题，实际上就是一个决策树的遍历过程**。你只需要思考 3 个问题：

1、路径：也就是已经做出的选择。

2、选择列表：也就是你当前可以做的选择。

3、结束条件：也就是到达决策树底层，无法再做选择的条件。

如果你不理解这三个词语的解释，没关系，我们后面会用「全排列」和「N 皇后问题」这两个经典的回溯算法问题来帮你理解这些词语是什么意思，现在你先留着印象。

代码方面，回溯算法的框架：

```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return
```

```
    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

其核心就是 for 循环里面的递归，在递归调用之前「做选择」，在递归调用之后「撤销选择」，特别简单。

什么叫做选择和撤销选择呢，这个框架的底层原理是什么呢？下面我们就通过「全排列」这个问题来解开之前的疑惑，详细探究一下其中的奥妙！

一、全排列问题

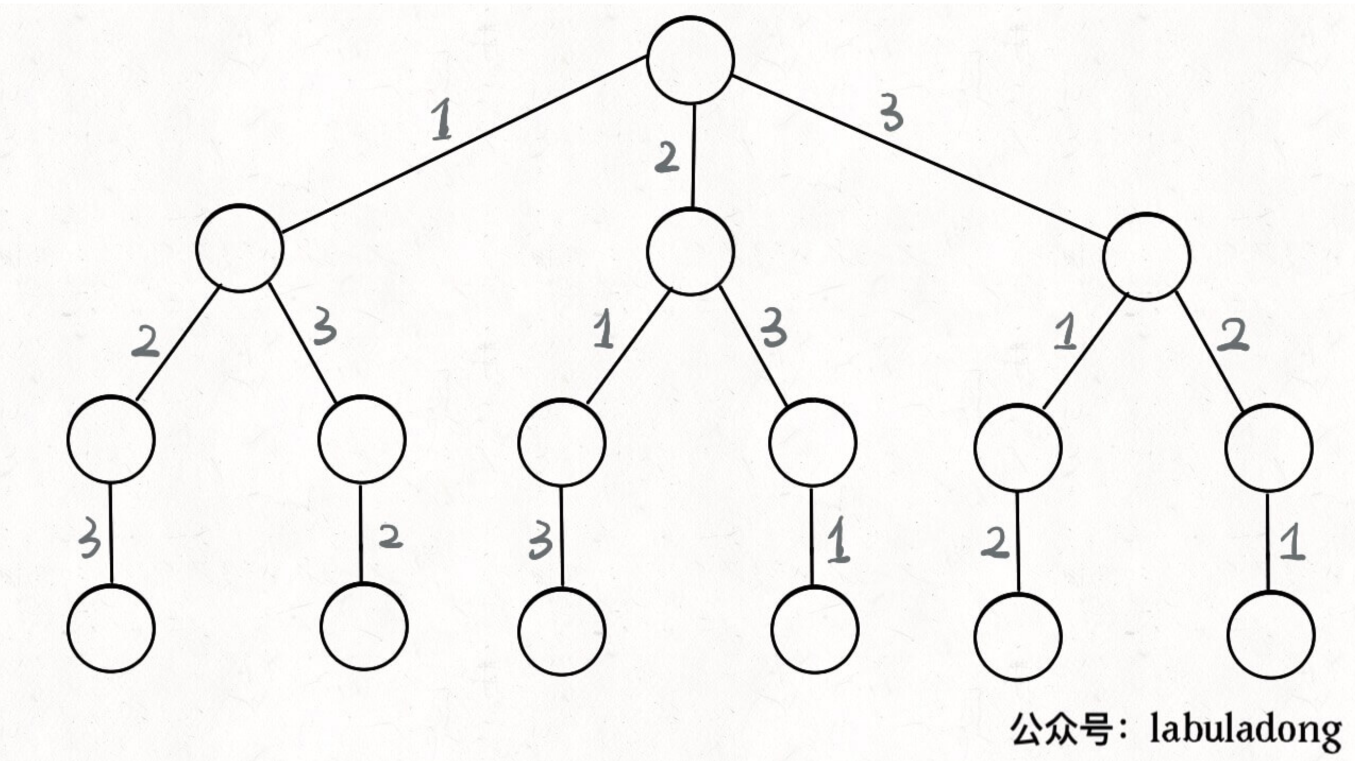
力扣第 46 题「全排列」就是这个问题，不过我们在高中的时候就做过排列组合的数学题，我们也知道 `n` 个不重复的数，全排列共有 `n!` 个。

> PS：**为了简单清晰起见，我们这次讨论的全排列问题不包含重复的数字。**

那么我们当时是怎么穷举全排列的呢？比方说给三个数 `[1,2,3]`，你肯定不会无规律地乱穷举，一般是这样：

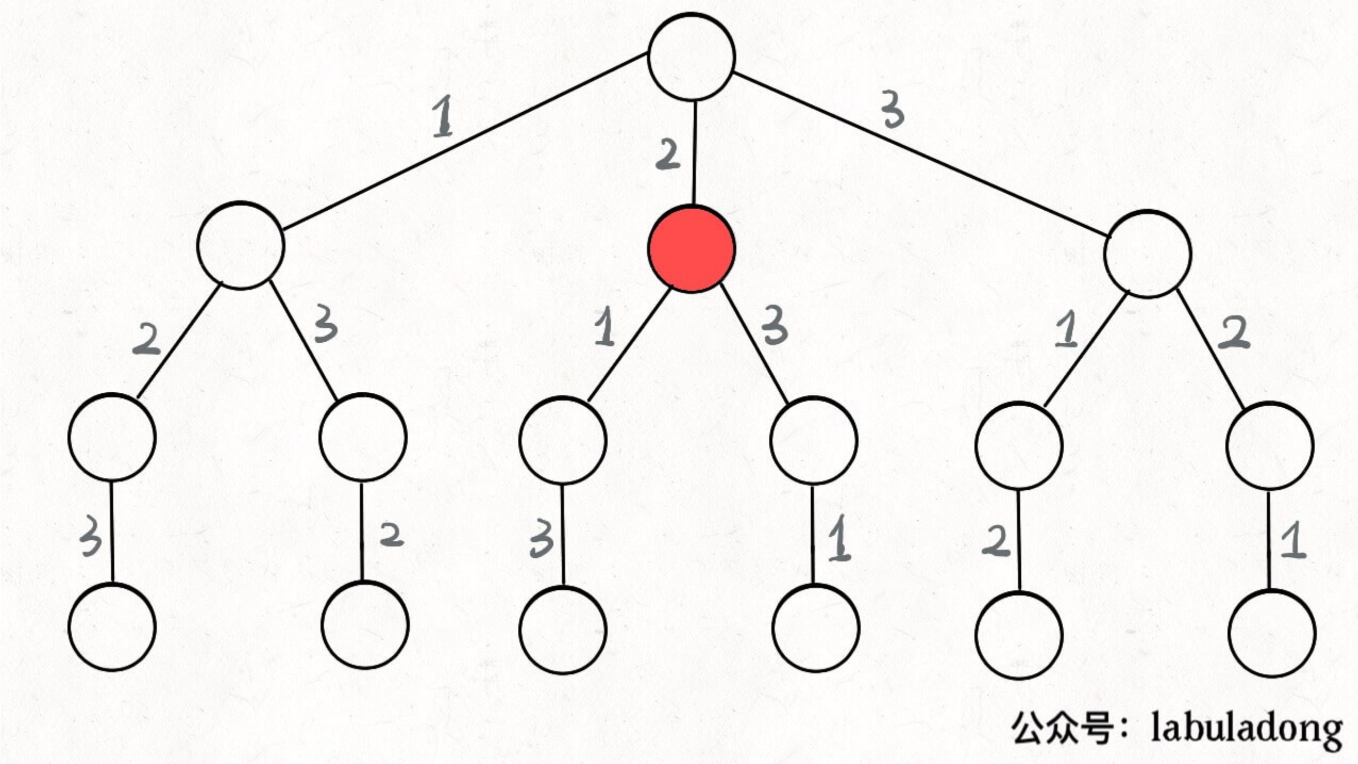
先固定第一位为 1，然后第二位可以是 2，那么第三位只能是 3；然后可以把第二位变成 3，第三位就只能是 2 了；然后就只能变化第一位，变成 2，然后再穷举后两位……

其实这就是回溯算法，我们高中无师自通就会用，或者有的同学直接画出如下这棵回溯树：



只要从根遍历这棵树，记录路径上的数字，其实就是所有的全排列。**我们不妨把这棵树称为回溯算法的「决策树」**。

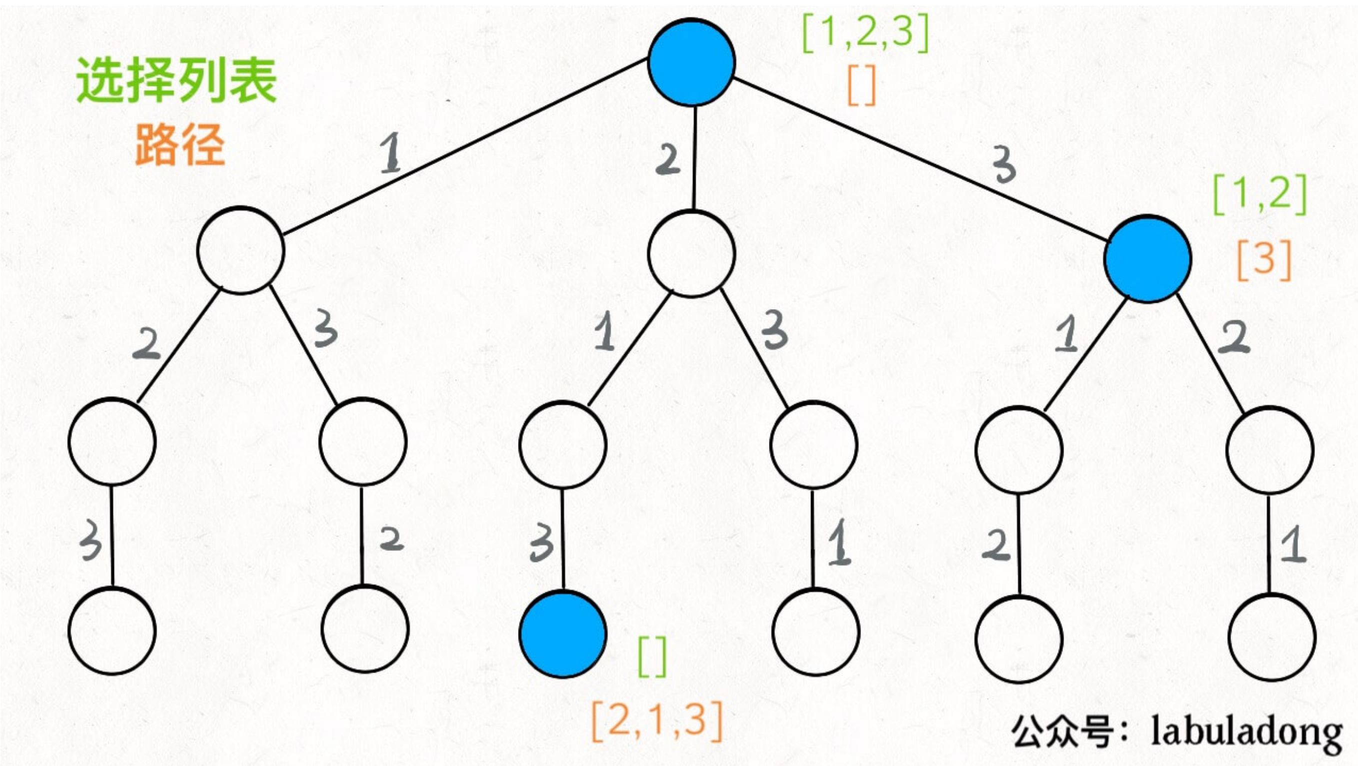
为啥说这是决策树呢，因为你在每个节点上其实都在做决策。比如说你站在下图的红色节点上：



你现在就在做决策，可以选择 1 那条树枝，也可以选择 3 那条树枝。为啥只能在 1 和 3 之中选择呢？因为 2 这个树枝在你身后，这个选择你之前做过了，而全排列是不允许重复使用数字的。

现在可以解答开头的几个名词：`[2]` 就是「路径」，记录你已经做过的选择；`[1,3]` 就是「选择列表」，表示你当前可以做出的选择；「结束条件」就是遍历到树的底层，在这里就是选择列表为空的时候。

如果明白了这几个名词，**可以把「路径」和「选择」列表作为决策树上每个节点的属性**，比如下图列出了几个节点的属性：

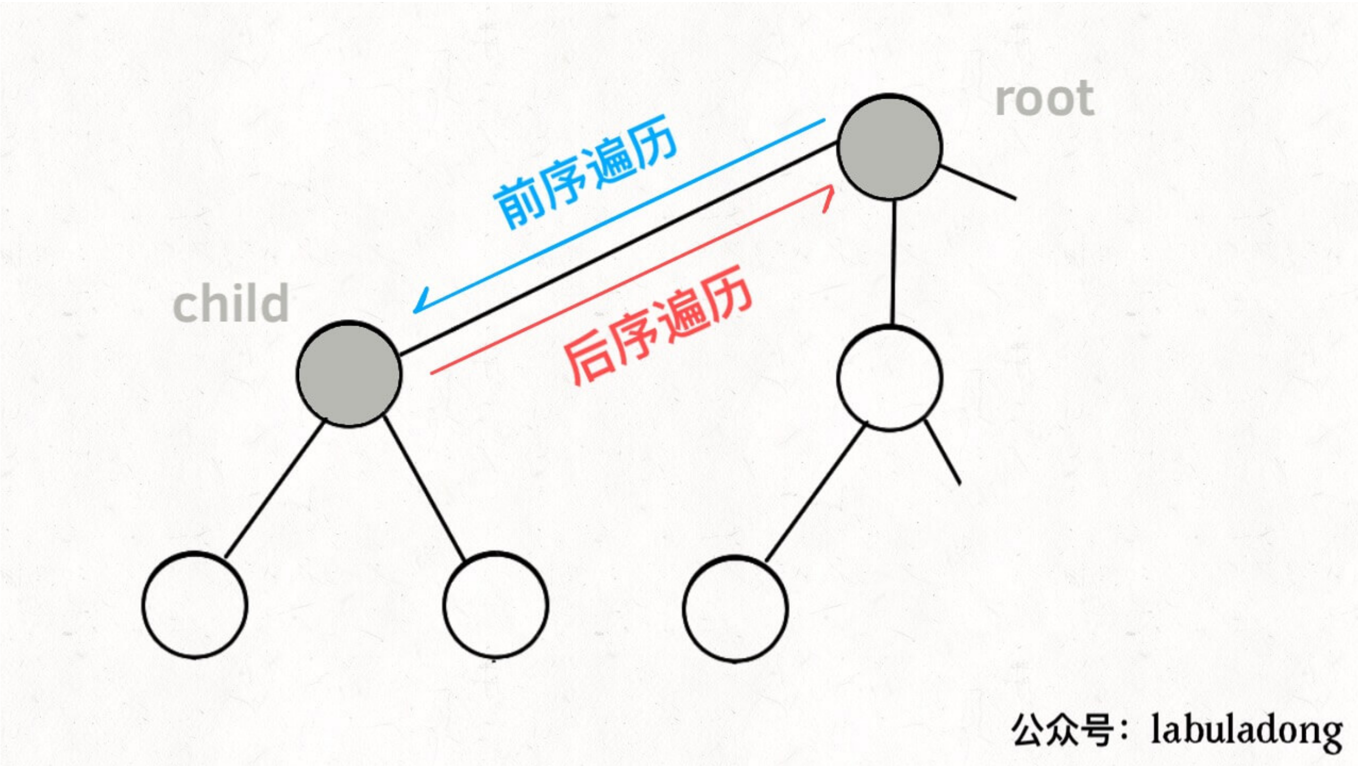


我们定义的 `backtrack` 函数其实就像一个指针，在这棵树上游走，同时要正确维护每个节点的属性，每当走到树的底层，其「路径」就是一个全排列。

再进一步，如何遍历一棵树？这个应该不难吧。回忆一下之前 学习数据结构的框架思维 写过，各种搜索问题其实都是树的遍历问题，而多叉树的遍历框架就是这样：

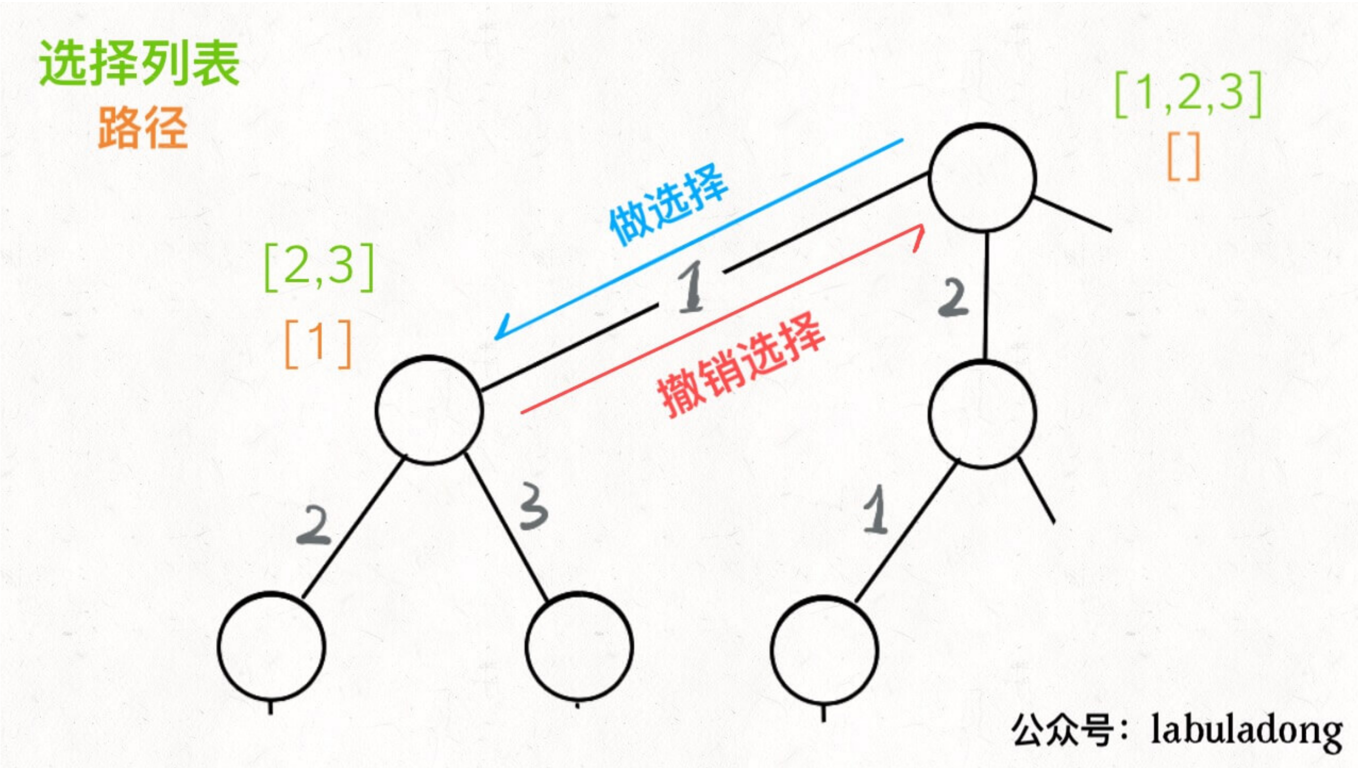
```
void traverse(TreeNode root) {
    for (TreeNode child : root.childern)
        // 前序遍历需要的操作
        traverse(child);
        // 后序遍历需要的操作
}
```

而所谓的前序遍历和后序遍历，他们只是两个很有用的时间点，我给你画张图你就明白了：



前序遍历的代码在进入某一个节点之前的那个时间点执行，后序遍历代码在离开某个节点之后的那个时间点执行。

回想我们刚才说的，「路径」和「选择」是每个节点的属性，函数在树上游走要正确维护节点的属性，那么就要在这两个特殊时间点搞点动作：



现在，你是否理解了回溯算法的这段核心框架？

```
for 选择 in 选择列表:
    # 做选择
    将该选择从选择列表移除
    路径.add(选择)
    backtrack(路径, 选择列表)
    # 撤销选择
    路径.remove(选择)
    将该选择再加入选择列表
```

我们只要在递归之前做出选择，在递归之后撤销刚才的选择，就能正确得到每个节点的选择列表和路径。

下面，直接看全排列代码：

```
List<List<Integer>> res = new LinkedList<>();

/* 主函数，输入一组不重复的数字，返回它们的全排列 */
List<List<Integer>> permute(int[] nums) {
    // 记录「路径」
    LinkedList<Integer> track = new LinkedList<>();
    // 「路径」中的元素会被标记为 true，避免重复使用
    boolean[] used = new boolean[nums.length];

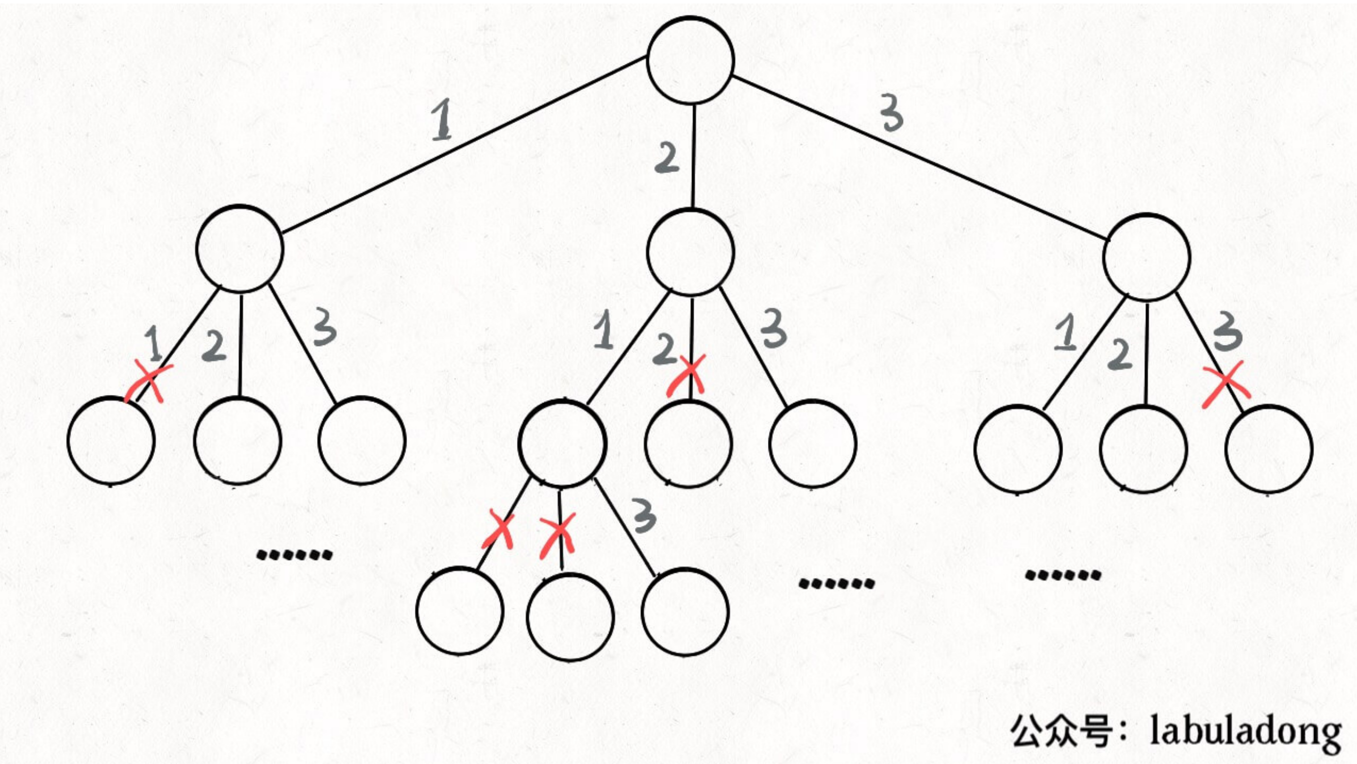
    backtrack(nums, track, used);
    return res;
}

// 路径：记录在 track 中
```

```java
    // 选择列表：nums 中不存在于 track 的那些元素（used[i] 为 false）
    // 结束条件：nums 中的元素全都在 track 中出现
    void backtrack(int[] nums, LinkedList<Integer> track, boolean[] used) {
        // 触发结束条件
        if (track.size() == nums.length) {
            res.add(new LinkedList(track));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            // 排除不合法的选择
            if (used[i]) {
                // nums[i] 已经在 track 中，跳过
                continue;
            }
            // 做选择
            track.add(nums[i]);
            used[i] = true;
            // 进入下一层决策树
            backtrack(nums, track, used);
            // 取消选择
            track.removeLast();
            used[i] = false;
        }
    }
```

我们这里稍微做了些变通，没有显式记录「选择列表」，而是通过 used 数组记录 track 中的元素，从而推导出当前的选择列表：



至此，我们就通过全排列问题详解了回溯算法的底层原理。当然，这个算法解决全排列不是最高效的，你可能看到有的解法连 used 数组都不使用，通过交换元素达到目的。但是那种解法稍微难理解一些，这里就不

写了，有兴趣可以自行搜索一下。

但是必须说明的是，不管怎么优化，都符合回溯框架，而且时间复杂度都不可能低于 O(N!)，因为穷举整棵决策树是无法避免的。**这也是回溯算法的一个特点，不像动态规划存在重叠子问题可以优化，回溯算法就是纯暴力穷举，复杂度一般都很高。**

明白了全排列问题，就可以直接套回溯算法框架了，下面简单看看 N 皇后问题。

二、N 皇后问题

力扣第 51 题「N 皇后」就是这个经典问题，简单解释一下：给你一个 N×N 的棋盘，让你放置 N 个皇后，使得它们不能互相攻击。

> PS：皇后可以攻击同一行、同一列、左上左下右上右下四个方向的任意单位。

这个问题本质上跟全排列问题差不多，决策树的每一层表示棋盘上的每一行；每个节点可以做出的选择是，在该行的任意一列放置一个皇后。

因为 C++ 代码对字符串的操作方便一些，所以这道题我用 C++ 来写解法，直接套用回溯算法框架:

```
vector<vector<string>> res;

/* 输入棋盘边长 n，返回所有合法的放置 */
vector<vector<string>> solveNQueens(int n) {
    // '.' 表示空，'Q' 表示皇后，初始化空棋盘。
    vector<string> board(n, string(n, '.'));
    backtrack(board, 0);
    return res;
}

// 路径：board 中小于 row 的那些行都已经成功放置了皇后
// 选择列表：第 row 行的所有列都是放置皇后的选择
// 结束条件：row 超过 board 的最后一行
void backtrack(vector<string>& board, int row) {
    // 触发结束条件
    if (row == board.size()) {
        res.push_back(board);
        return;
    }

    int n = board[row].size();
    for (int col = 0; col < n; col++) {
        // 排除不合法选择
        if (!isValid(board, row, col)) {
            continue;
        }
        // 做选择
        board[row][col] = 'Q';
        // 进入下一行决策
        backtrack(board, row + 1);
        // 撤销选择
        board[row][col] = '.';
```

```
    }
}
```

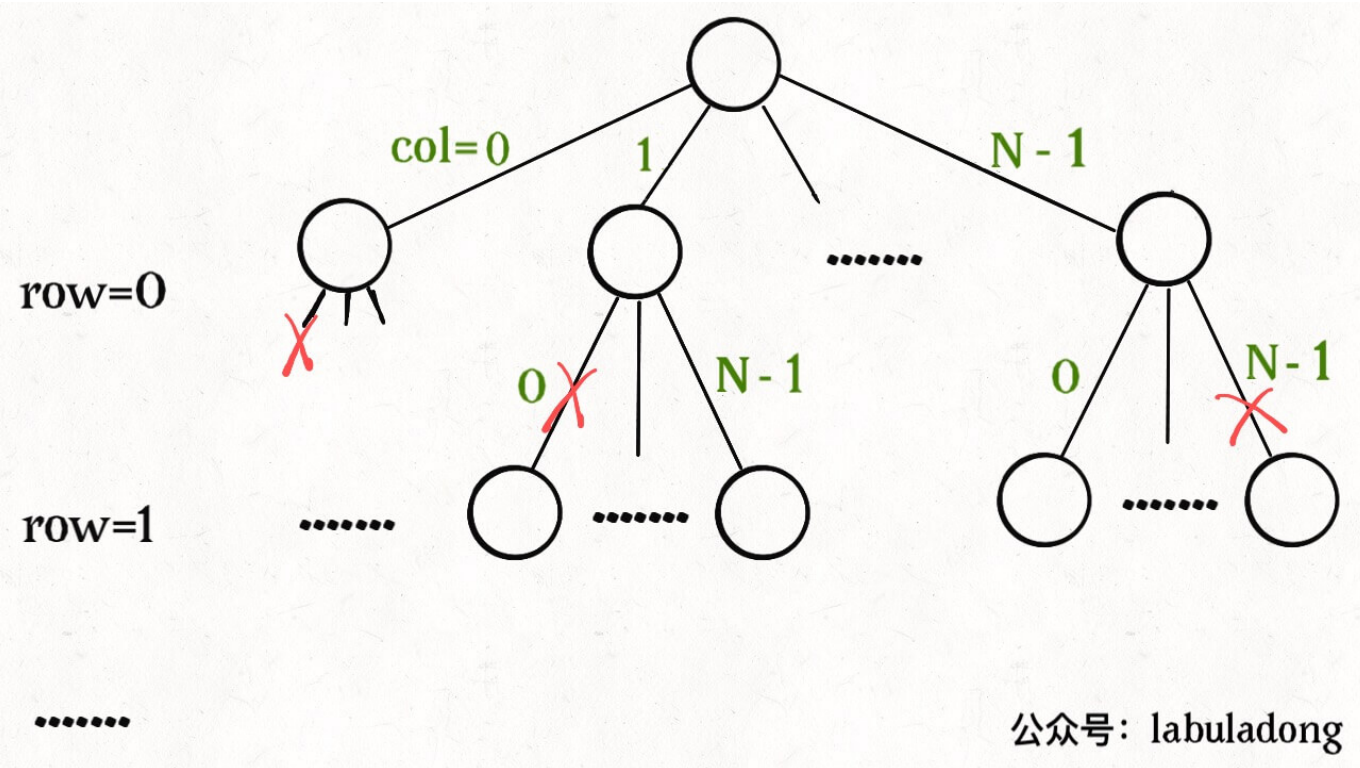
这部分主要代码，其实跟全排列问题差不多，`isValid` 函数的实现也很简单：

```
/* 是否可以在 board[row][col] 放置皇后？ */
bool isValid(vector<string>& board, int row, int col) {
    int n = board.size();
    // 检查列是否有皇后互相冲突
    for (int i = 0; i < n; i++) {
        if (board[i][col] == 'Q')
            return false;
    }
    // 检查右上方是否有皇后互相冲突
    for (int i = row - 1, j = col + 1;
            i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 'Q')
            return false;
    }
    // 检查左上方是否有皇后互相冲突
    for (int i = row - 1, j = col - 1;
            i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 'Q')
            return false;
    }
    return true;
}
```

> PS：肯定有读者问，按照 N 皇后问题的描述，我们为什么不检查左下角，右下角和下方的格子，只检查了左上角，右上角和上方的格子呢？

因为皇后是一行一行从上往下放的，所以左下方，右下方和正下方不用检查（还没放皇后）；因为一行只会放一个皇后，所以每行不用检查。也就是最后只用检查上面，左上，右上三个方向。

函数 `backtrack` 依然像个在决策树上游走的指针，通过 `row` 和 `col` 就可以表示函数遍历到的位置，通过 `isValid` 函数可以将不符合条件的情况剪枝：



如果直接给你这么一大段解法代码，可能是懵逼的。但是现在明白了回溯算法的框架套路，还有啥难理解的呢？无非是改改做选择的方式，排除不合法选择的方式而已，只要框架存于心，你面对的只剩下小问题了。

当 `N = 8` 时，就是八皇后问题，数学大佬高斯穷尽一生都没有数清楚八皇后问题到底有几种可能的放置方法，但是我们的算法只需要一秒就可以算出来所有可能的结果。

不过真的不怪高斯。这个问题的复杂度确实非常高，看看我们的决策树，虽然有 `isValid` 函数剪枝，但是最坏时间复杂度仍然是 O(N^(N+1))，而且无法优化。如果 `N = 10` 的时候，计算就已经很耗时了。

有的时候，我们并不想得到所有合法的答案，只想要一个答案，怎么办呢？比如解数独的算法，找所有解法复杂度太高，只要找到一种解法就可以。

其实特别简单，只要稍微修改一下回溯算法的代码即可：

```
// 函数找到一个答案后就返回 true
bool backtrack(vector<string>& board, int row) {
    // 触发结束条件
    if (row == board.size()) {
        res.push_back(board);
        return true;
    }
    ...
    for (int col = 0; col < n; col++) {
        ...
        board[row][col] = 'Q';

        if (backtrack(board, row + 1))
            return true;

        board[row][col] = '.';
    }
```

```
        return false;
    }
```

这样修改后，只要找到一个答案，for 循环的后续递归穷举都会被阻断。也许你可以在 N 皇后问题的代码框架上，稍加修改，写一个解数独的算法？

三、最后总结

回溯算法就是个多叉树的遍历问题，关键就是在前序遍历和后序遍历的位置做一些操作，算法框架如下：

```python
def backtrack(...):
    for 选择 in 选择列表:
        做选择
        backtrack(...)
        撤销选择
```

写 `backtrack` 函数时，需要维护走过的「路径」和当前可以做的「选择列表」，当触发「结束条件」时，将「路径」记入结果集。

其实想想看，回溯算法和动态规划是不是有点像呢？我们在动态规划系列文章中多次强调，动态规划的三个需要明确的点就是「状态」「选择」和「base case」，是不是就对应着走过的「路径」，当前的「选择列表」和「结束条件」？

某种程度上说，动态规划的暴力求解阶段就是回溯算法。只是有的问题具有重叠子问题性质，可以用 dp table 或者备忘录优化，将递归树大幅剪枝，这就变成了动态规划。而今天的两个问题，都没有重叠子问题，也就是回溯算法问题了，复杂度非常高是不可避免的。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

＿＿＿＿＿＿＿＿＿＿＿＿＿

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



回溯算法牛逼：集合划分问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong



学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

698. 划分为k个相等的子集（中等）

之前说过回溯算法是笔试中最好用的算法，只要你没什么思路，就用回溯算法暴力求解，即便不能通过所有测试用例，多少能过一点。

回溯算法的技巧也不难，前文 回溯算法框架套路 说过，回溯算法就是穷举一棵决策树的过程，只要在递归之前「做选择」，在递归之后「撤销选择」就行了。

但是，就算暴力穷举，不同的思路也有优劣之分。

本文就来看一道非常经典的回溯算法问题，力扣第 698 题「划分为k个相等的子集」。这道题可以帮你更深刻理解回溯算法的思维，得心应手地写出回溯函数。

题目非常简单：

给你输入一个数组 `nums` 和一个正整数 `k`，请你判断 `nums` 是否能够被平分为元素和相同的 `k` 个子集。

函数签名如下：

```
boolean canPartitionKSubsets(int[] nums, int k);
```

我们之前 背包问题之子集划分 写过一次子集划分问题，不过那道题只需要我们把集合划分成两个相等的集合，可以转化成背包问题用动态规划技巧解决。

但是如果划分成多个相等的集合，解法一般只能通过暴力穷举，时间复杂度爆表，是练习回溯算法和递归思维的好机会。

一、思路分析

首先，我们回顾一下以前学过的排列组合知识：

1、`P(n, k)`（也有很多书写成 `A(n, k)`）表示从 `n` 个不同元素中拿出 `k` 个元素的排列（Permutation/Arrangement）；`C(n, k)` 表示从 `n` 个不同元素中拿出 `k` 个元素的组合（Combination）总

数。

2、「排列」和「组合」的主要区别在于是否考虑顺序的差异。

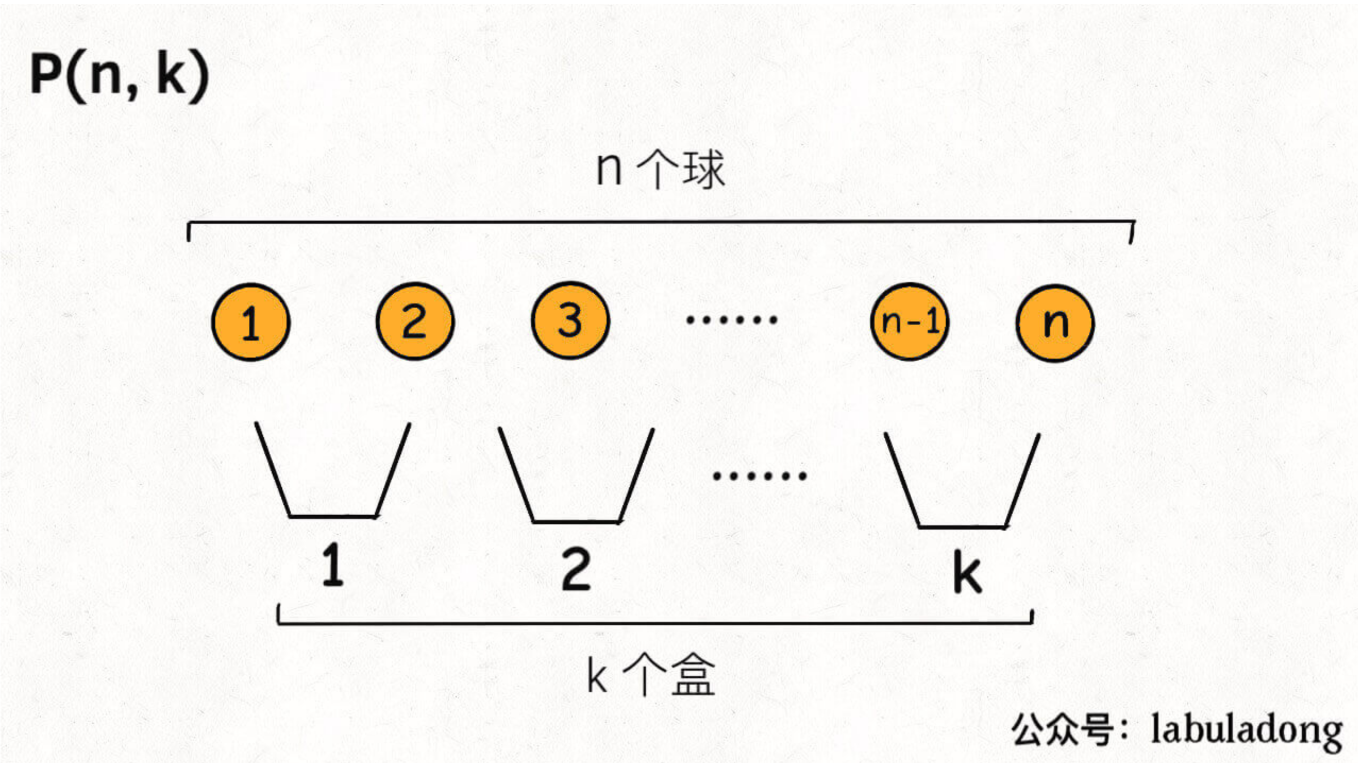
3、排列、组合总数的计算公式：

$$P(n, k) = \frac{n!}{(n-k)!}$$

$$C(n, k) = \frac{n!}{k!(n-k)!}$$

好，现在我问一个问题，这个排列公式 P(n, k) 是如何推导出来的？为了搞清楚这个问题，我需要讲一点组合数学的知识。

排列组合问题的各种变体都可以抽象成「球盒模型」，P(n, k) 就可以抽象成下面这个场景：

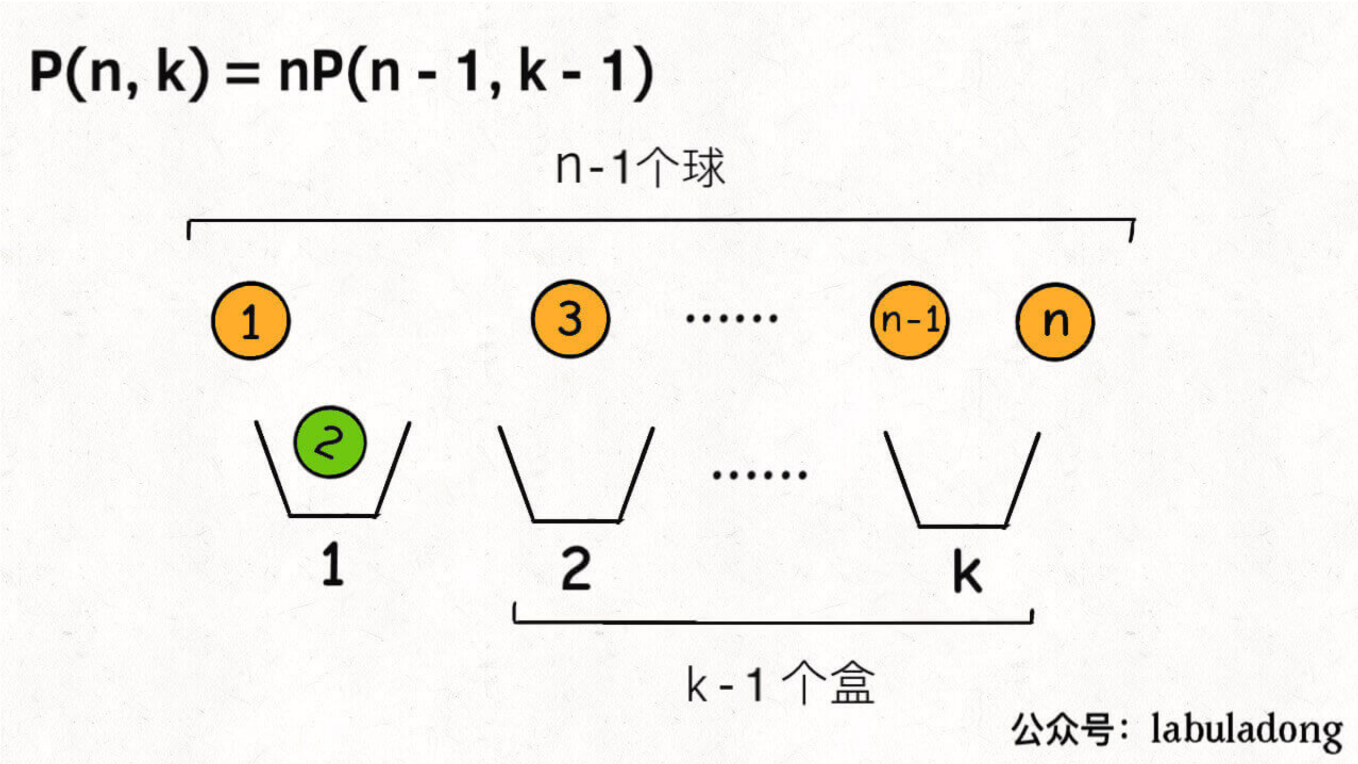


即，将 n 个标记了不同序号的球（标号为了体现顺序的差异），放入 k 个标记了不同序号的盒子中（其中 n >= k，每个盒子最终都装有恰好一个球），共有 P(n, k) 种不同的方法。

现在你来，往盒子里放球，你怎么放？其实有两种视角。

首先，你可以站在盒子的视角，每个盒子必然要选择一个球。

这样，第一个盒子可以选择 n 个球中的任意一个，然后你需要让剩下 k - 1 个盒子在 n - 1 个球中选择：

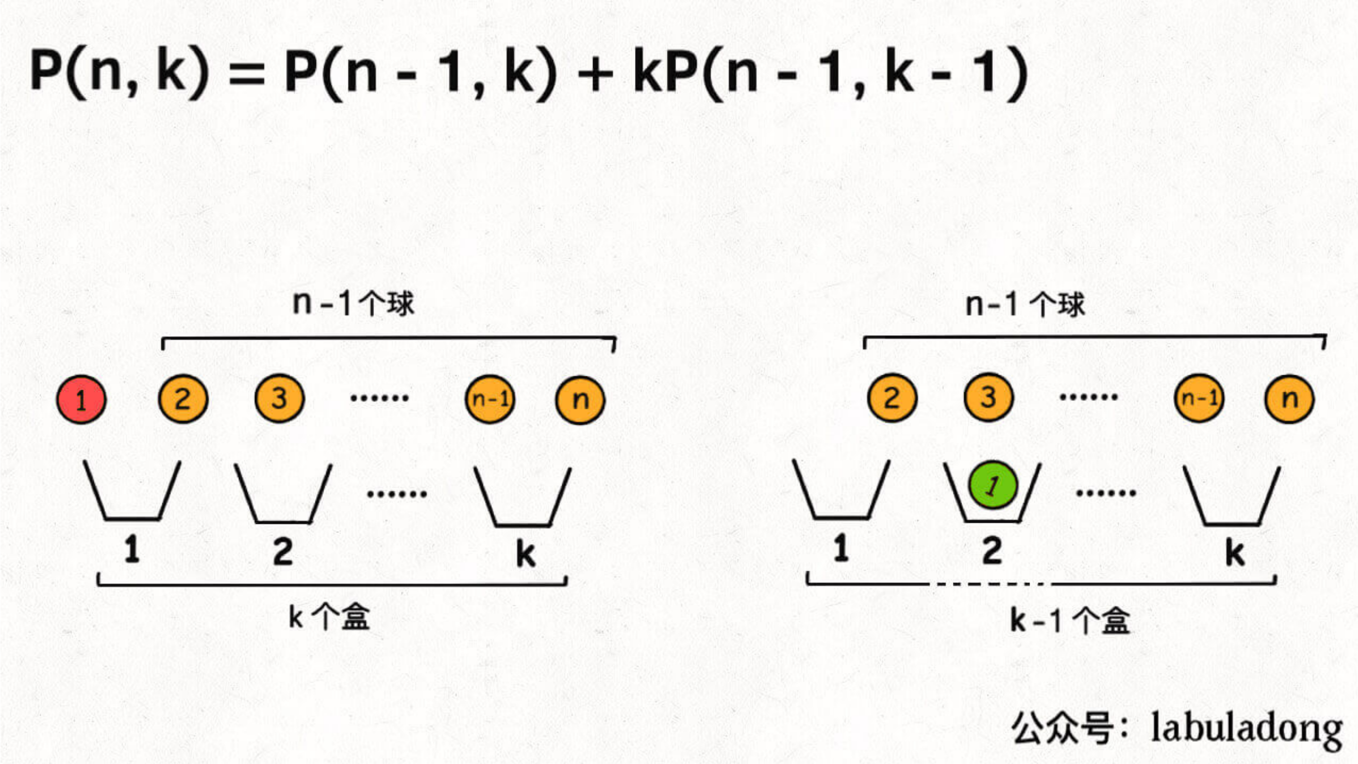


另外，你也可以站在球的视角，因为并不是每个球都会被装进盒子，所以球的视角分两种情况：

1、第一个球可以不装进任何一个盒子，这样的话你就需要将剩下 n - 1 个球放入 k 个盒子。

2、第一个球可以装进 k 个盒子中的任意一个，这样的话你就需要将剩下 n - 1 个球放入 k - 1 个盒子。

结合上述两种情况，可以得到：



你看，两种视角得到两个不同的递归式，但这两个递归式解开的结果都是我们熟知的阶乘形式：

$$
\begin{aligned}
& P(n, k) \\
& = nP(n-1, k-1) \\
& = P(n-1, k) + kP(n-1, k-1) \\
& = \frac{n!}{(n-k)!}
\end{aligned}
$$

至于如何解递归式，涉及数学的内容比较多，这里就不做深入探讨了，有兴趣的读者可以自行学习组合数学相关知识。

回到正题，这道算法题让我们求子集划分，子集问题和排列组合问题有所区别，但我们可以借鉴「球盒模型」的抽象，用两种不同的视角来解决这道子集划分问题。

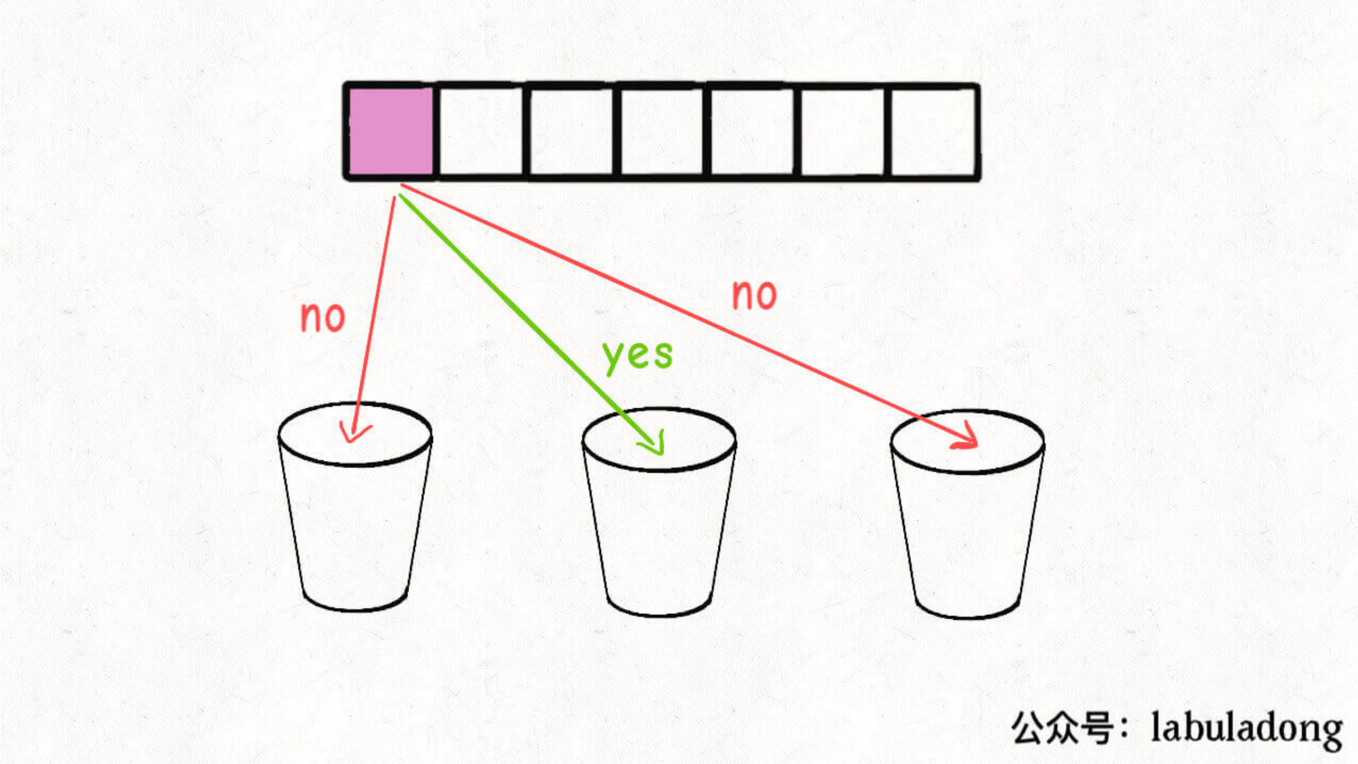
把装有 `n` 个数字的数组 `nums` 分成 `k` 个和相同的集合，你可以想象将 `n` 个数字分配到 `k` 个「桶」里，最后这 `k` 个「桶」里的数字之和要相同。

前文 回溯算法框架套路 说过，回溯算法的关键在哪里？

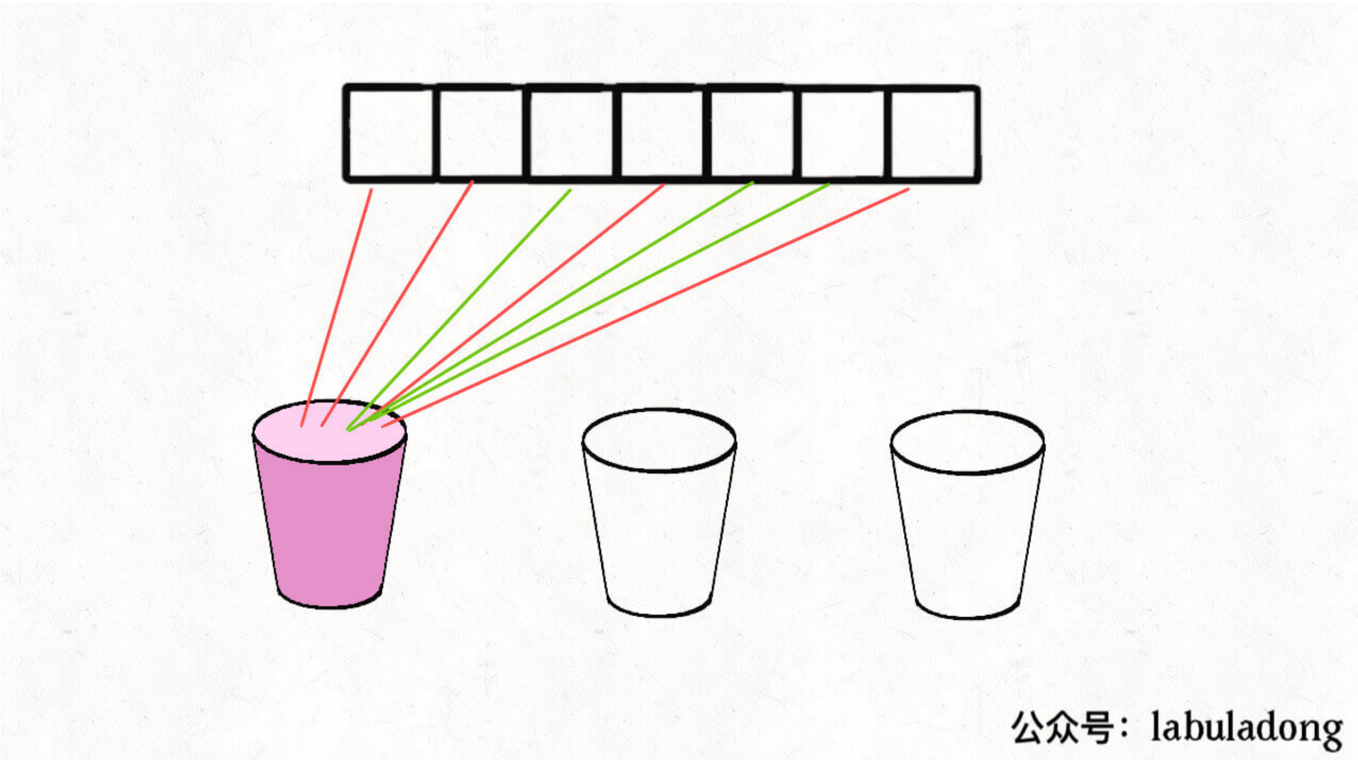
关键是要知道怎么「做选择」，这样才能利用递归函数进行穷举。

那么模仿排列公式的推导思路，将 `n` 个数字分配到 `k` 个桶里，我们也可以有两种视角：

视角一，如果我们切换到这 `n` 个数字的视角，每个数字都要选择进入到 `k` 个桶中的某一个。



视角二，如果我们切换到这 `k` 个桶的视角，对于每个桶，都要遍历 `nums` 中的 `n` 个数字，然后选择是否将当前遍历到的数字装进自己这个桶里。



你可能问，这两种视角有什么不同？

用不同的视角进行穷举，虽然结果相同，但是解法代码的逻辑完全不同，进而算法的效率也会不同；对比不同的穷举视角，可以帮你更深刻地理解回溯算法，我们慢慢道来。

二、以数字的视角

用 for 循环迭代遍历 `nums` 数组大家肯定都会：

```
for (int index = 0; index < nums.length; index++) {
    System.out.println(nums[index]);
}
```

递归遍历数组你会不会？其实也很简单：

```
void traverse(int[] nums, int index) {
    if (index == nums.length) {
        return;
    }
    System.out.println(nums[index]);
    traverse(nums, index + 1);
}
```

只要调用 `traverse(nums, 0)`，和 for 循环的效果是完全一样的。

那么回到这道题，以数字的视角，选择 `k` 个桶，用 for 循环写出来是下面这样：

```
// k 个桶（集合），记录每个桶装的数字之和
int[] bucket = new int[k];

// 穷举 nums 中的每个数字
for (int index = 0; index < nums.length; index++) {
    // 穷举每个桶
    for (int i = 0; i < k; i++) {
        // nums[index] 选择是否要进入第 i 个桶
        // ...
    }
}
```

如果改成递归的形式，就是下面这段代码逻辑：

```
// k 个桶（集合），记录每个桶装的数字之和
int[] bucket = new int[k];

// 穷举 nums 中的每个数字
void backtrack(int[] nums, int index) {
    // base case
    if (index == nums.length) {
        return;
    }
    // 穷举每个桶
    for (int i = 0; i < bucket.length; i++) {
        // 选择装进第 i 个桶
        bucket[i] += nums[index];
        // 递归穷举下一个数字的选择
```

```java
            backtrack(nums, index + 1);
            // 撤销选择
            bucket[i] -= nums[index];
        }
    }
```

虽然上述代码仅仅是穷举逻辑，还不能解决我们的问题，但是只要略加完善即可：

```java
// 主函数
boolean canPartitionKSubsets(int[] nums, int k) {
    // 排除一些基本情况
    if (k > nums.length) return false;
    int sum = 0;
    for (int v : nums) sum += v;
    if (sum % k != 0) return false;

    // k 个桶（集合），记录每个桶装的数字之和
    int[] bucket = new int[k];
    // 理论上每个桶（集合）中数字的和
    int target = sum / k;
    // 穷举，看看 nums 是否能划分成 k 个和为 target 的子集
    return backtrack(nums, 0, bucket, target);
}

// 递归穷举 nums 中的每个数字
boolean backtrack(
    int[] nums, int index, int[] bucket, int target) {

    if (index == nums.length) {
        // 检查所有桶的数字之和是否都是 target
        for (int i = 0; i < bucket.length; i++) {
            if (bucket[i] != target) {
                return false;
            }
        }
        // nums 成功平分成 k 个子集
        return true;
    }

    // 穷举 nums[index] 可能装入的桶
    for (int i = 0; i < bucket.length; i++) {
        // 剪枝，桶装装满了
        if (bucket[i] + nums[index] > target) {
            continue;
        }
        // 将 nums[index] 装入 bucket[i]
        bucket[i] += nums[index];
        // 递归穷举下一个数字的选择
        if (backtrack(nums, index + 1, bucket, target)) {
            return true;
        }
        // 撤销选择
```

```
            bucket[i] -= nums[index];
        }

        // nums[index] 装入哪个桶都不行
        return false;
    }
```

有之前的铺垫，相信这段代码是比较容易理解的。这个解法虽然能够通过，但是耗时比较多，其实我们可以再做一个优化。

主要看 `backtrack` 函数的递归部分：

```
for (int i = 0; i < bucket.length; i++) {
    // 剪枝
    if (bucket[i] + nums[index] > target) {
        continue;
    }

    if (backtrack(nums, index + 1, bucket, target)) {
        return true;
    }
}
```

如果我们让尽可能多的情况命中剪枝的那个 if 分支，就可以减少递归调用的次数，一定程度上减少时间复杂度。

如何尽可能多的命中这个 if 分支呢？要知道我们的 `index` 参数是从 0 开始递增的，也就是递归地从 0 开始遍历 `nums` 数组。

如果我们提前对 `nums` 数组排序，把大的数字排在前面，那么大的数字会先被分配到 `bucket` 中，对于之后的数字，`bucket[i] + nums[index]` 会更大，更容易触发剪枝的 if 条件。

所以可以在之前的代码中再添加一些代码：

```
boolean canPartitionKSubsets(int[] nums, int k) {
    // 其他代码不变
    // ...
    /* 降序排序 nums 数组 */
    Arrays.sort(nums);
    for (i = 0, j = nums.length - 1; i < j; i++, j--) {
        // 交换 nums[i] 和 nums[j]
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
    /*******************/
    return backtrack(nums, 0, bucket, target);
}
```

由于 Java 的语言特性，这段代码通过先升序排序再反转，达到降序排列的目的。

三、以桶的视角

文章开头说了，**以桶的视角进行穷举，每个桶需要遍历 `nums` 中的所有数字，决定是否把当前数字装进桶中；当装满一个桶之后，还要装下一个桶，直到所有桶都装满为止。**

这个思路可以用下面这段代码表示出来：

```
// 装满所有桶为止
while (k > 0) {
    // 记录当前桶中的数字之和
    int bucket = 0;
    for (int i = 0; i < nums.length; i++) {
        // 决定是否将 nums[i] 放入当前桶中
        bucket += nums[i] or 0;
        if (bucket == target) {
            // 装满了一个桶，装下一个桶
            k--;
            break;
        }
    }
}
```

那么我们也可以把这个 while 循环改写成递归函数，不过比刚才略微复杂一些，首先写一个 `backtrack` 递归函数出来：

```
boolean backtrack(int k, int bucket,
    int[] nums, int start, boolean[] used, int target);
```

不要被这么多参数吓到，我会一个个解释这些参数。**如果你能够透彻理解本文，也能得心应手地写出这样的回溯函数。**

这个 `backtrack` 函数的参数可以这样解释：

现在 `k` 号桶正在思考是否应该把 `nums[start]` 这个元素装进来；目前 `k` 号桶里面已经装的数字之和为 `bucket`；`used` 标志某一个元素是否已经被装到桶中；`target` 是每个桶需要达成的目标和。

根据这个函数定义，可以这样调用 `backtrack` 函数：

```
boolean canPartitionKSubsets(int[] nums, int k) {
    // 排除一些基本情况
    if (k > nums.length) return false;
    int sum = 0;
    for (int v : nums) sum += v;
    if (sum % k != 0) return false;

    boolean[] used = new boolean[nums.length];
```

```
    int target = sum / k;
    // k 号桶初始什么都没装，从 nums[0] 开始做选择
    return backtrack(k, 0, nums, 0, used, target);
}
```

实现 backtrack 函数的逻辑之前，再重复一遍，从桶的视角：

1、需要遍历 nums 中所有数字，决定哪些数字需要装到当前桶中。

2、如果当前桶装满了（桶内数字和达到 target），则让下一个桶开始执行第 1 步。

下面的代码就实现了这个逻辑：

```
boolean backtrack(int k, int bucket,
    int[] nums, int start, boolean[] used, int target) {
    // base case
    if (k == 0) {
        // 所有桶都被装满了，而且 nums 一定全部用完了
        // 因为 target == sum / k
        return true;
    }
    if (bucket == target) {
        // 装满了当前桶，递归穷举下一个桶的选择
        // 让下一个桶从 nums[0] 开始选数字
        return backtrack(k - 1, 0 ,nums, 0, used, target);
    }

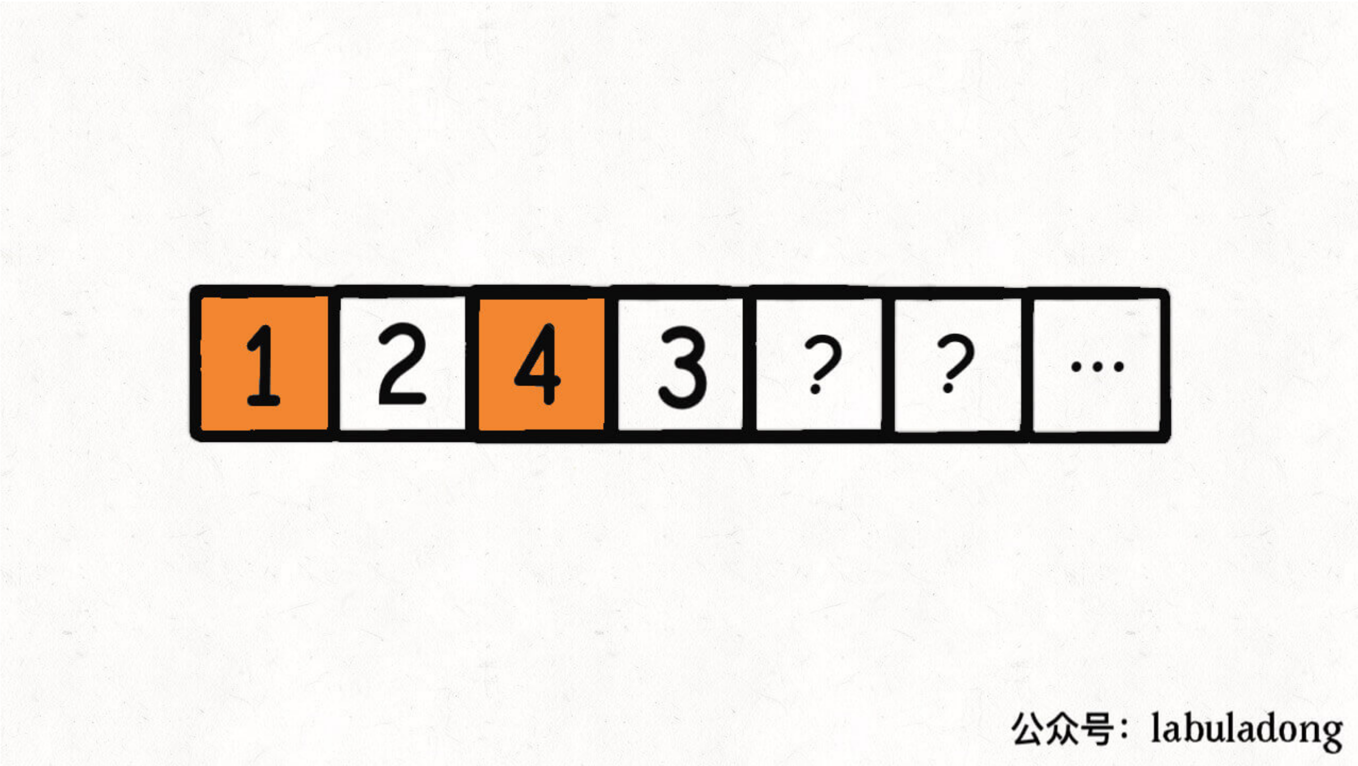
    // 从 start 开始向后探查有效的 nums[i] 装入当前桶
    for (int i = start; i < nums.length; i++) {
        // 剪枝
        if (used[i]) {
            // nums[i] 已经被装入别的桶中
            continue;
        }
        if (nums[i] + bucket > target) {
            // 当前桶装不下 nums[i]
            continue;
        }
        // 做选择，将 nums[i] 装入当前桶中
        used[i] = true;
        bucket += nums[i];
        // 递归穷举下一个数字是否装入当前桶
        if (backtrack(k, bucket, nums, i + 1, used, target)) {
            return true;
        }
        // 撤销选择
        used[i] = false;
        bucket -= nums[i];
    }
    // 穷举了所有数字，都无法装满当前桶
    return false;
}
```

这段代码是可以得出正确答案的，但是效率很低，我们可以思考一下是否还有优化的空间。

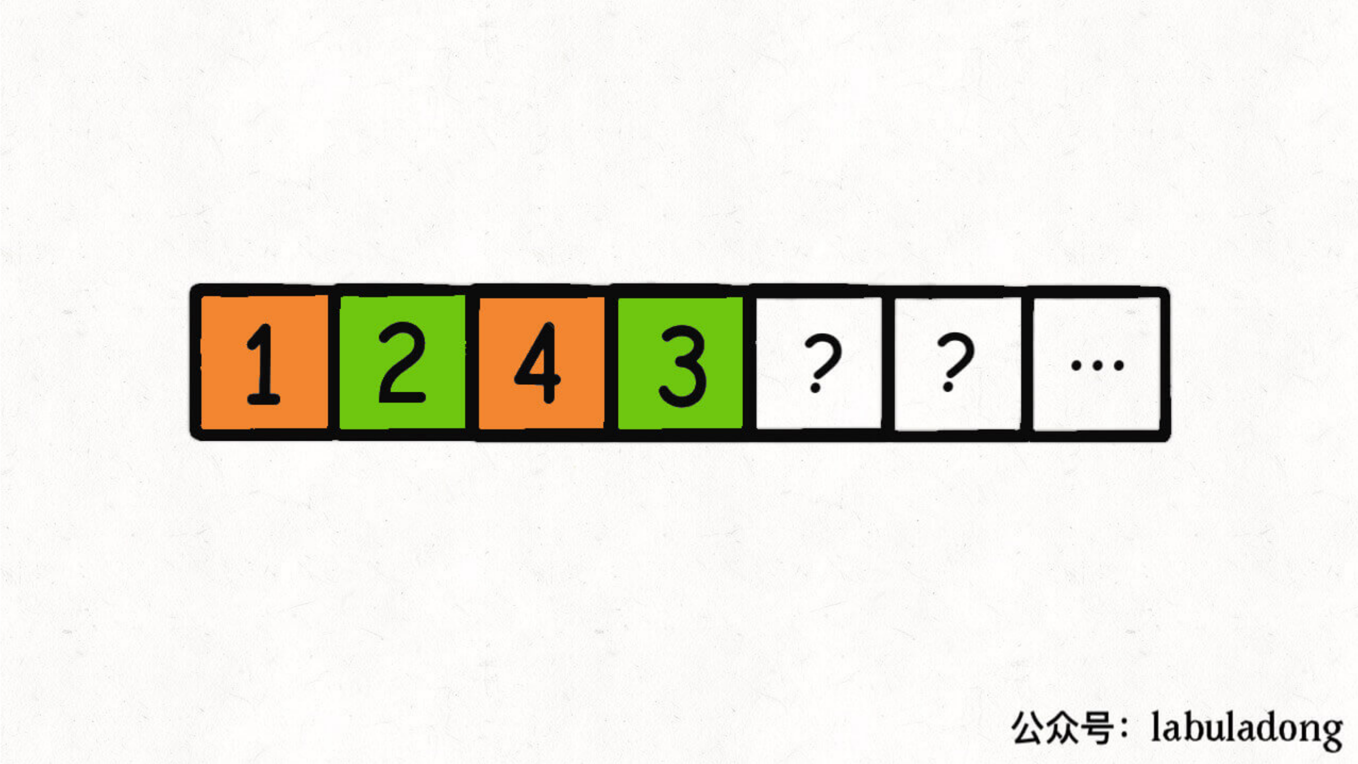
首先，在这个解法中每个桶都可以认为是没有差异的，但是我们的回溯算法却会对它们区别对待，这里就会出现重复计算的情况。

什么意思呢？我们的回溯算法，说到底就是穷举所有可能的组合，然后看是否能找出和为 `target` 的 `k` 个桶（子集）。

那么，比如下面这种情况，`target = 5`，算法会在第一个桶里面装 `1, 4`：



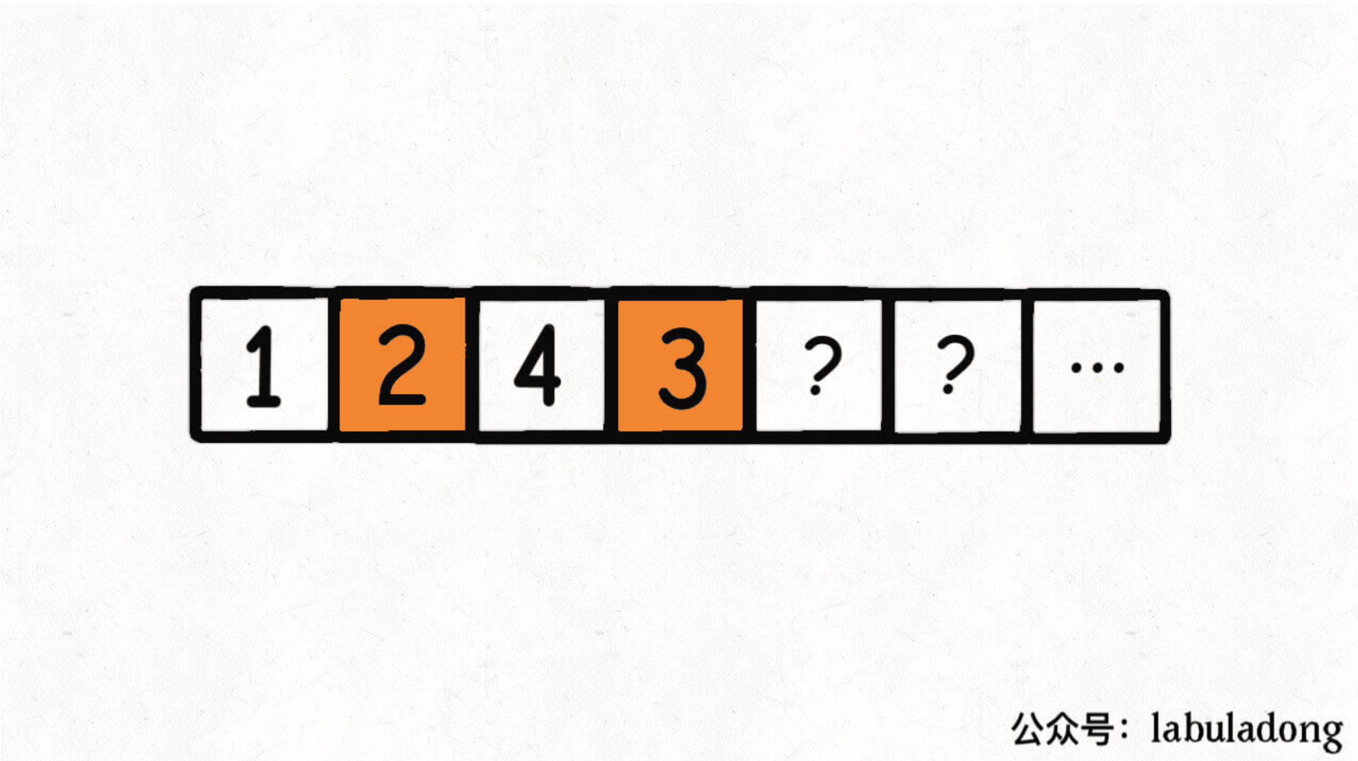
现在第一个桶装满了，就开始装第二个桶，算法会装入 `2, 3`：



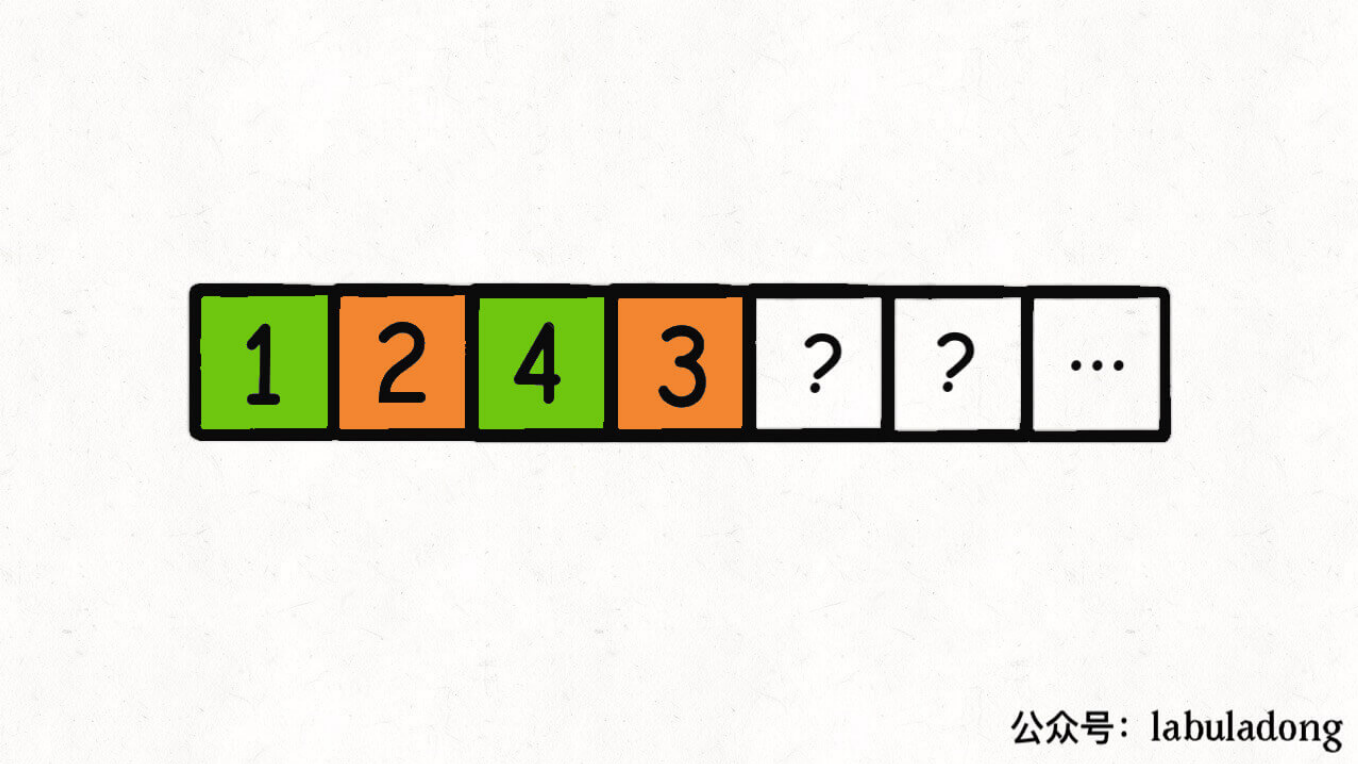
然后以此类推，对后面的元素进行穷举，凑出若干个和为 5 的桶（子集）。

但问题是，如果最后发现无法凑出和为 `target` 的 `k` 个子集，算法会怎么做？

回溯算法会回溯到第一个桶，重新开始穷举，现在它知道第一个桶里装 `1, 4` 是不可行的，它会尝试把 `2, 3` 装到第一个桶里：



现在第一个桶装满了，就开始装第二个桶，算法会装入 `1, 4`：



好，到这里你应该看出来问题了，这种情况其实和之前的那种情况是一样的。也就是说，到这里你其实已经知道不需要再穷举了，必然凑不出来和为 target 的 k 个子集。

但我们的算法还是会傻乎乎地继续穷举，因为在她看来，第一个桶和第二个桶里面装的元素不一样，那这就是两种不一样的情况呀。

那么我们怎么让算法的智商提高，识别出这种情况，避免冗余计算呢？

你注意这两种情况的 used 数组肯定长得一样，所以 used 数组可以认为是回溯过程中的「状态」。

所以，我们可以用一个 memo 备忘录，在装满一个桶时记录当前 used 的状态，如果当前 used 的状态是曾经出现过的，那就不用再继续穷举，从而起到剪枝避免冗余计算的作用。

有读者肯定会问，used 是一个布尔数组，怎么作为键进行存储呢？这其实是小问题，比如我们可以把数组转化成字符串，这样就可以作为哈希表的键进行存储了。

看下代码实现，只要稍微改一下 backtrack 函数即可：

```
// 备忘录，存储 used 数组的状态
HashMap<String, Boolean> memo = new HashMap<>();

boolean backtrack(int k, int bucket, int[] nums, int start, boolean[]
used, int target) {
    // base case
    if (k == 0) {
        return true;
    }
    // 将 used 的状态转化成形如 [true, false, ...] 的字符串
    // 便于存入 HashMap
    String state = Arrays.toString(used);
```

```
    if (bucket == target) {
        // 装满了当前桶，递归穷举下一个桶的选择
        boolean res = backtrack(k - 1, 0, nums, 0, used, target);
        // 将当前状态和结果存入备忘录
        memo.put(state, res);
        return res;
    }

    if (memo.containsKey(state)) {
        // 如果当前状态曾今计算过，就直接返回，不要再递归穷举了
        return memo.get(state);
    }

    // 其他逻辑不变...
}
```

这样提交解法，发现执行效率依然比较低，这次不是因为算法逻辑上的冗余计算，而是代码实现上的问题。

因为每次递归都要把 `used` 数组转化成字符串，这对于编程语言来说也是一个不小的消耗，所以我们还可以进一步优化。

注意题目给的数据规模 `nums.length <= 16`，也就是说 `used` 数组最多也不会超过 16，那么我们完全可以用「位图」的技巧，用一个 int 类型的 `used` 变量来替代 `used` 数组。

具体来说，我们可以用整数 `used` 的第 `i` 位（`(used >> i) & 1`）的 1/0 来表示 `used[i]` 的 true/false。

这样一来，不仅节约了空间，而且整数 `used` 也可以直接作为键存入 HashMap，省去数组转字符串的消耗。

看下最终的解法代码：

```
public boolean canPartitionKSubsets(int[] nums, int k) {
    // 排除一些基本情况
    if (k > nums.length) return false;
    int sum = 0;
    for (int v : nums) sum += v;
    if (sum % k != 0) return false;

    int used = 0; // 使用位图技巧
    int target = sum / k;
    // k 号桶初始什么都没装，从 nums[0] 开始做选择
    return backtrack(k, 0, nums, 0, used, target);
}

HashMap<Integer, Boolean> memo = new HashMap<>();

boolean backtrack(int k, int bucket,
                  int[] nums, int start, int used, int target) {
    // base case
    if (k == 0) {
        // 所有桶都被装满了，而且 nums 一定全部用完了
        return true;
    }
```

```java
        if (bucket == target) {
            // 装满了当前桶，递归穷举下一个桶的选择
            // 让下一个桶从 nums[0] 开始选数字
            boolean res = backtrack(k - 1, 0, nums, 0, used, target);
            // 缓存结果
            memo.put(used, res);
            return res;
        }

        if (memo.containsKey(used)) {
            // 避免冗余计算
            return memo.get(used);
        }

        for (int i = start; i < nums.length; i++) {
            // 剪枝
            if (((used >> i) & 1) == 1) { // 判断第 i 位是否是 1
                // nums[i] 已经被装入别的桶中
                continue;
            }
            if (nums[i] + bucket > target) {
                continue;
            }
            // 做选择
            used |= 1 << i; // 将第 i 位置为 1
            bucket += nums[i];
            // 递归穷举下一个数字是否装入当前桶
            if (backtrack(k, bucket, nums, i + 1, used, target)) {
                return true;
            }
            // 撤销选择
            used ^= 1 << i; // 使用异或运算将第 i 位恢复 0
            bucket -= nums[i];
        }

        return false;
    }
```

至此，这道题的第二种思路也完成了。

四、最后总结

本文写的这两种思路都可以算出正确答案，不过第一种解法即便经过了排序优化，也明显比第二种解法慢很多，这是为什么呢？

我们来分析一下这两个算法的时间复杂度，假设 `nums` 中的元素个数为 `n`。

先说第一个解法，也就是从数字的角度进行穷举，`n` 个数字，每个数字有 `k` 个桶可供选择，所以组合出的结果个数为 `k^n`，时间复杂度也就是 `O(k^n)`。

第二个解法，每个桶要遍历 `n` 个数字，对每个数字有「装入」或「不装入」两种选择，所以组合的结果有 `2^n` 种；而我们有 `k` 个桶，所以总的时间复杂度为 `O(k*2^n)`。

当然，这是对最坏复杂度上界的粗略估算，实际的复杂度肯定要好很多，毕竟我们添加了这么多剪枝逻辑。 不过，从复杂度的上界已经可以看出第一种思路要慢很多了。

所以，谁说回溯算法没有技巧性的？虽然回溯算法就是暴力穷举，但穷举也分聪明的穷举方式和低效的穷举方式，关键看你以谁的「视角」进行穷举。

通俗来说，我们应该尽量「少量多次」，就是说宁可多做几次选择，也不要给太大的选择空间；宁可「二选一」选 k 次，也不要 「k 选一」选一次。

好了，这道题我们从两种视角进行穷举，虽然代码量看起来多，但核心逻辑都是类似的，相信你通过本文能够更深刻地理解回溯算法。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



一文秒杀所有排列/组合/子集问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

78. 子集（中等）

90. 子集 II（中等）

77. 组合（中等）

39. 组合总和（中等）

40. 组合总和 II（中等）

216. 组合总和 III（中等）

46. 全排列（中等）

47. 全排列 II（中等）

虽然这几个问题是高中就学过的，但如果想编写算法决这几类问题，还是非常考验计算机思维的，本文就讲讲编程解决这几个问题的核心思路，以后再有什么变体，你也能手到擒来，以不变应万变。

无论是排列、组合还是子集问题，简单说无非就是让你从序列 `nums` 中以给定规则取若干元素，主要有以下几种变体：

形式一、元素无重不可复选，即 `nums` 中的元素都是唯一的，每个元素最多只能被使用一次，这也是最基本的形式。

以组合为例，如果输入 `nums = [2,3,6,7]`，和为 7 的组合应该只有 `[7]`。

形式二、元素可重不可复选，即 `nums` 中的元素可以存在重复，每个元素最多只能被使用一次。

以组合为例，如果输入 `nums = [2,5,2,1,2]`，和为 7 的组合应该有两种 `[2,2,2,1]` 和 `[5,2]`。

形式三、元素无重可复选，即 `nums` 中的元素都是唯一的，每个元素可以被使用若干次。

以组合为例，如果输入 `nums = [2,3,6,7]`，和为 7 的组合应该有两种 `[2,2,3]` 和 `[7]`。

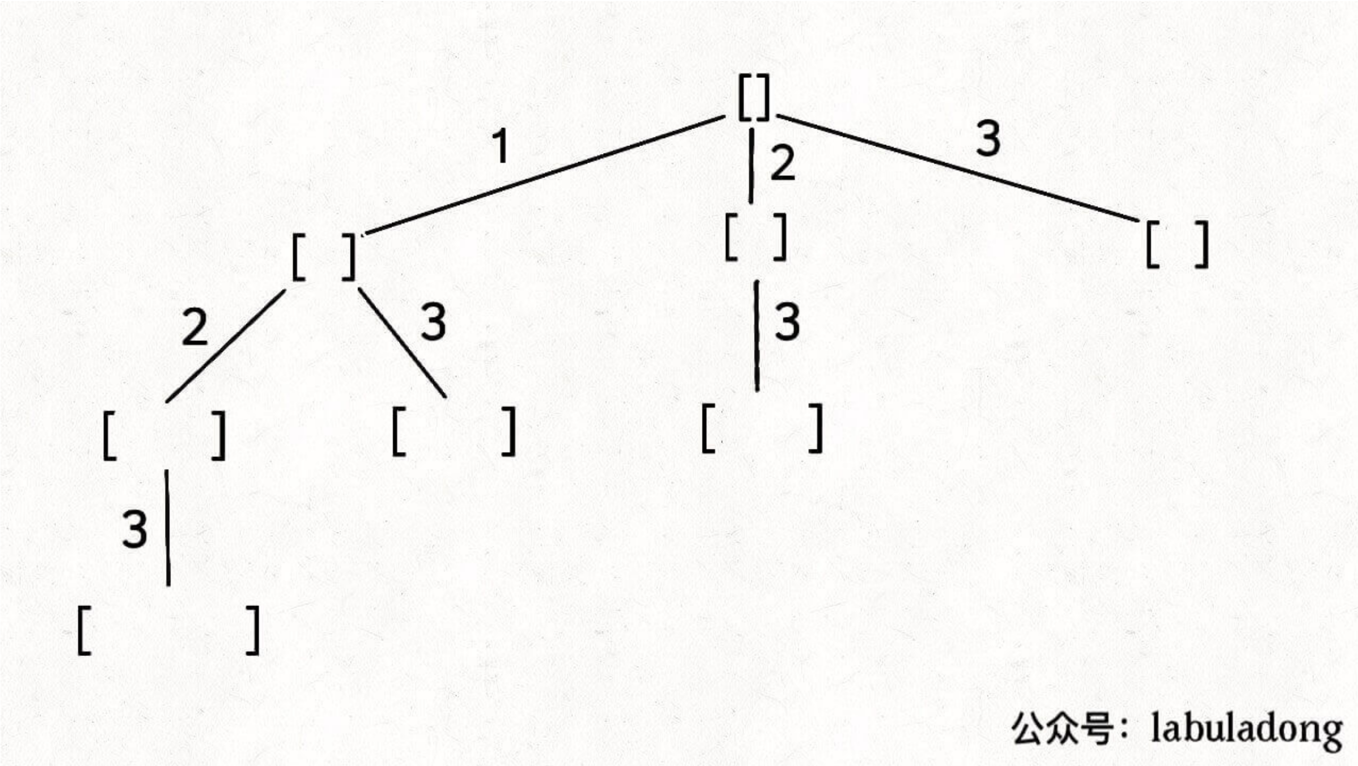
当然，也可以说有第四种形式，即元素可重可复选。但既然元素可复选，那又何必存在重复元素呢？元素去重之后就等同于形式三，所以这种情况不用考虑。

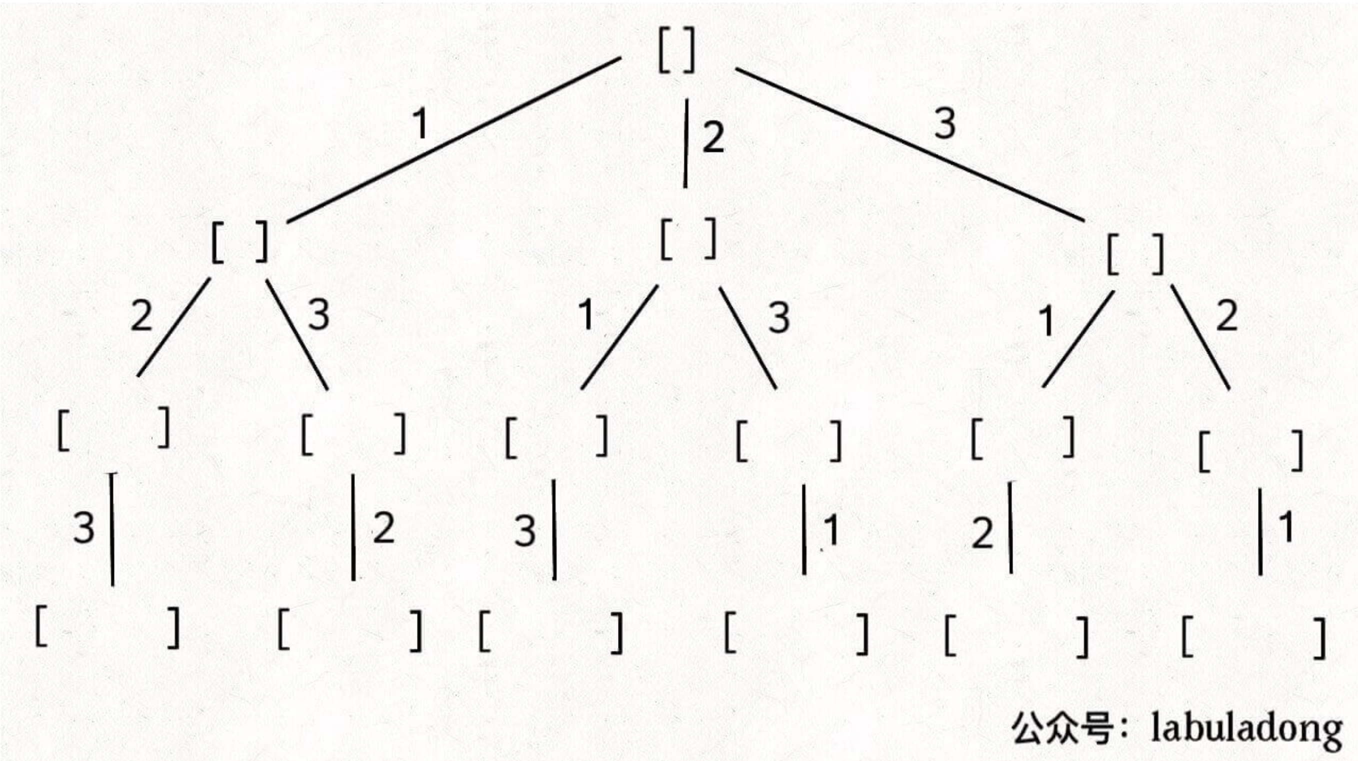
上面用组合问题举的例子，但排列、组合、子集问题都可以有这三种基本形式，所以共有 9 种变化。

除此之外，题目也可以再添加各种限制条件，比如让你求和为 `target` 且元素个数为 `k` 的组合，那这么一来又可以衍生出一堆变体，怪不得面试笔试中经常考到排列组合这种基本题型。

但无论形式怎么变化，其本质就是穷举所有解，而这些解呈现树形结构，所以合理使用回溯算法框架，稍改代码框架即可把这些问题一网打尽。

具体来说，你需要先阅读并理解前文 回溯算法核心套路，然后记住如下子集问题和排列问题的回溯树，就可以解决所有排列组合子集相关的问题：





为什么只要记住这两种树形结构就能解决所有相关问题呢？

首先，组合问题和子集问题其实是等价的，这个后面会讲；至于之前说的三种变化形式，无非是在这两棵树上剪掉或者增加一些树枝罢了。

那么，接下来我们就开始穷举，把排列/组合/子集问题的 9 种形式都过一遍，学学如何用回溯算法把它们一套带走。

子集（元素无重不可复选）

力扣第 78 题「子集」就是这个问题：

题目给你输入一个无重复元素的数组 `nums`，其中每个元素最多使用一次，请你返回 `nums` 的所有子集。

函数签名如下：

```
List<List<Integer>> subsets(int[] nums)
```

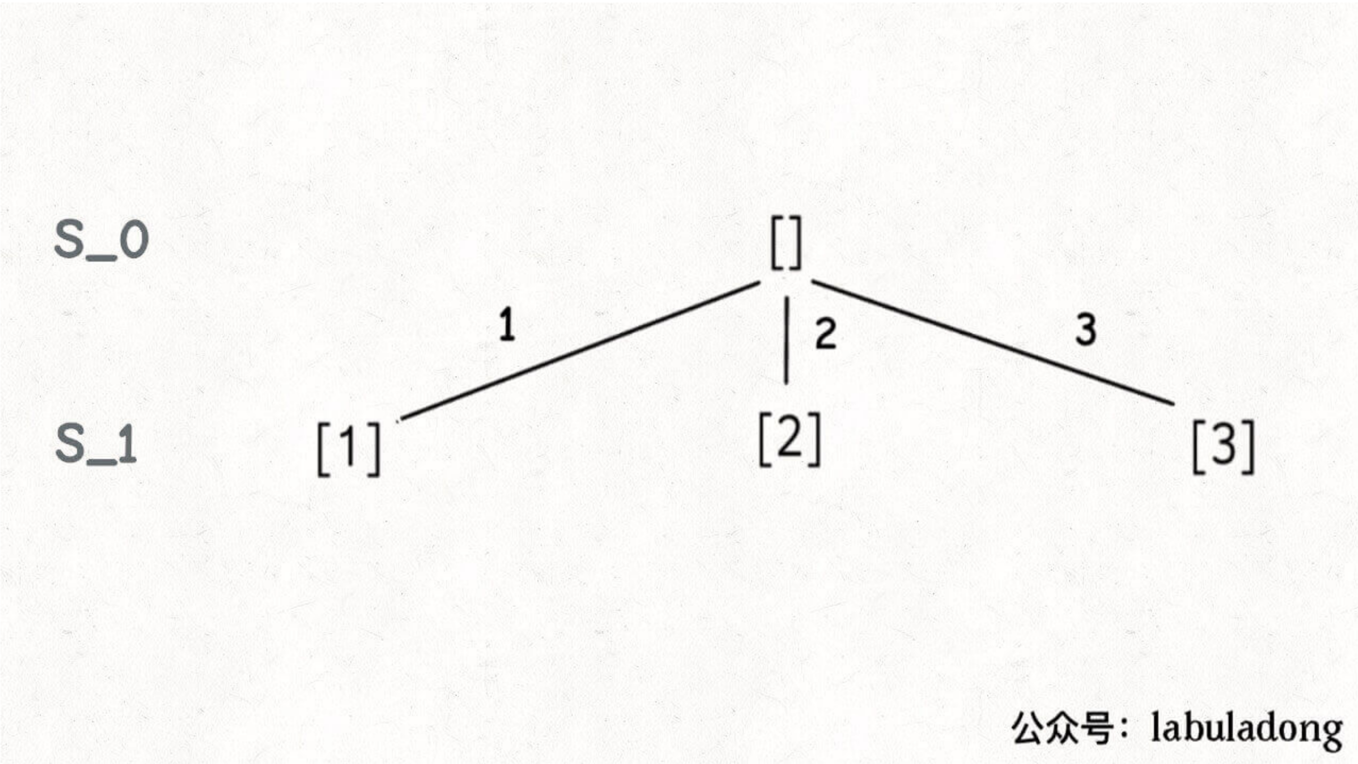
比如输入 `nums = [1,2,3]`，算法应该返回如下子集：

```
[ [],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3] ]
```

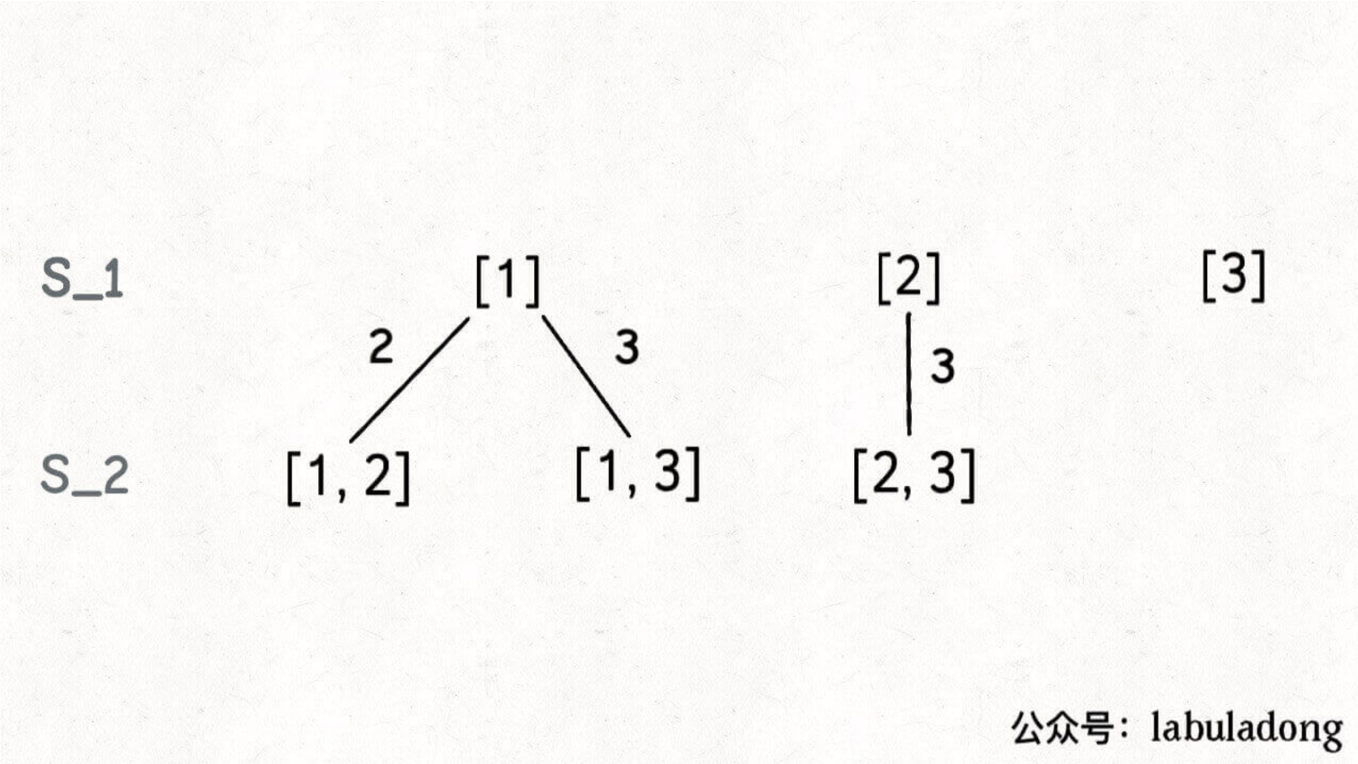
好，我们暂时不考虑如何用代码实现，先回忆一下我们的高中知识，如何手推所有子集？

首先，生成元素个数为 0 的子集，即空集 `[]`，为了方便表示，我称之为 `S_0`。

然后，在 `S_0` 的基础上生成元素个数为 1 的所有子集，我称为 `S_1`：



接下来，我们可以在 S_1 的基础上推导出 S_2，即元素个数为 2 的所有子集：



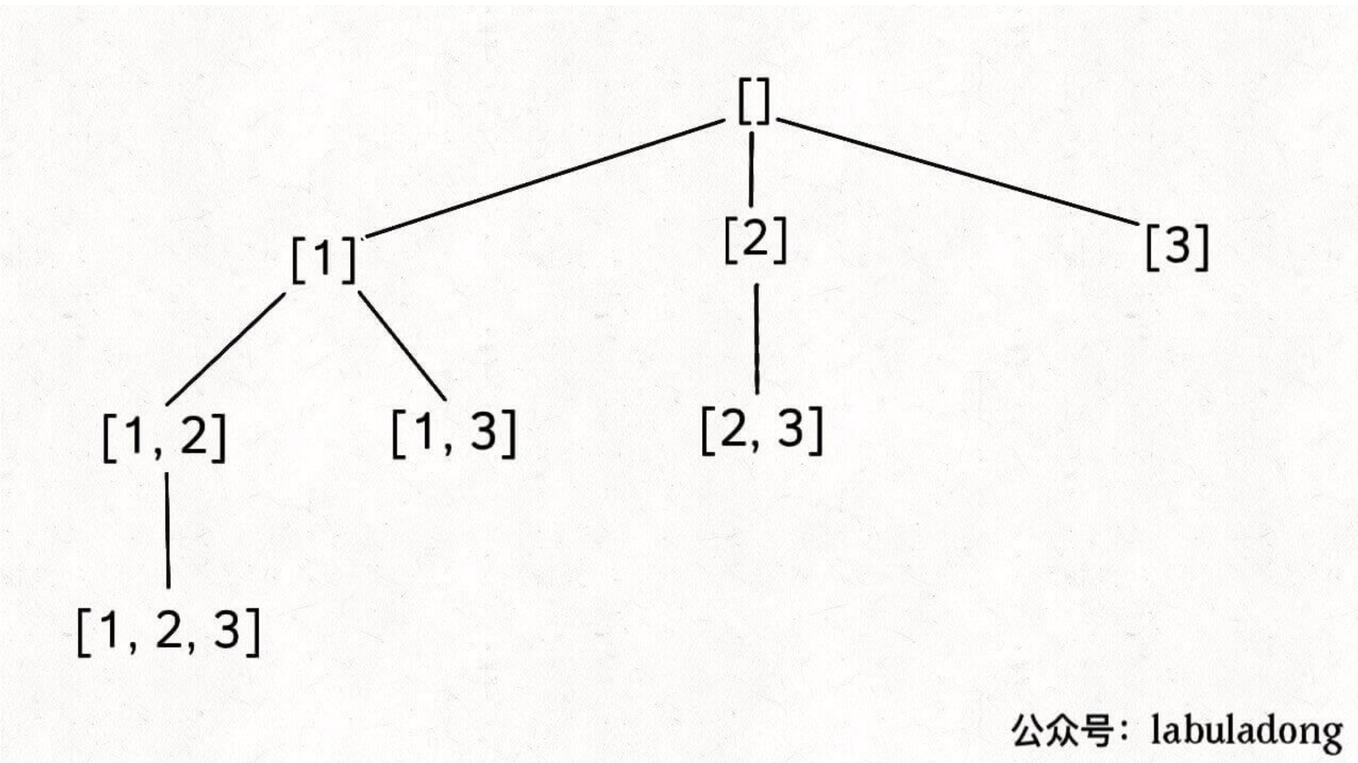
为什么集合 [2] 只需要添加 3，而不添加前面的 1 呢？

因为集合中的元素不用考虑顺序， [1,2,3] 中 2 后面只有 3，如果你向前考虑 1，那么 [2,1] 会和之前已经生成的子集 [1,2] 重复。

换句话说，我们通过保证元素之间的相对顺序不变来防止出现重复的子集。

接着，我们可以通过 S_2 推出 S_3，实际上 S_3 中只有一个集合 [1,2,3]，它是通过 [1,2] 推出的。

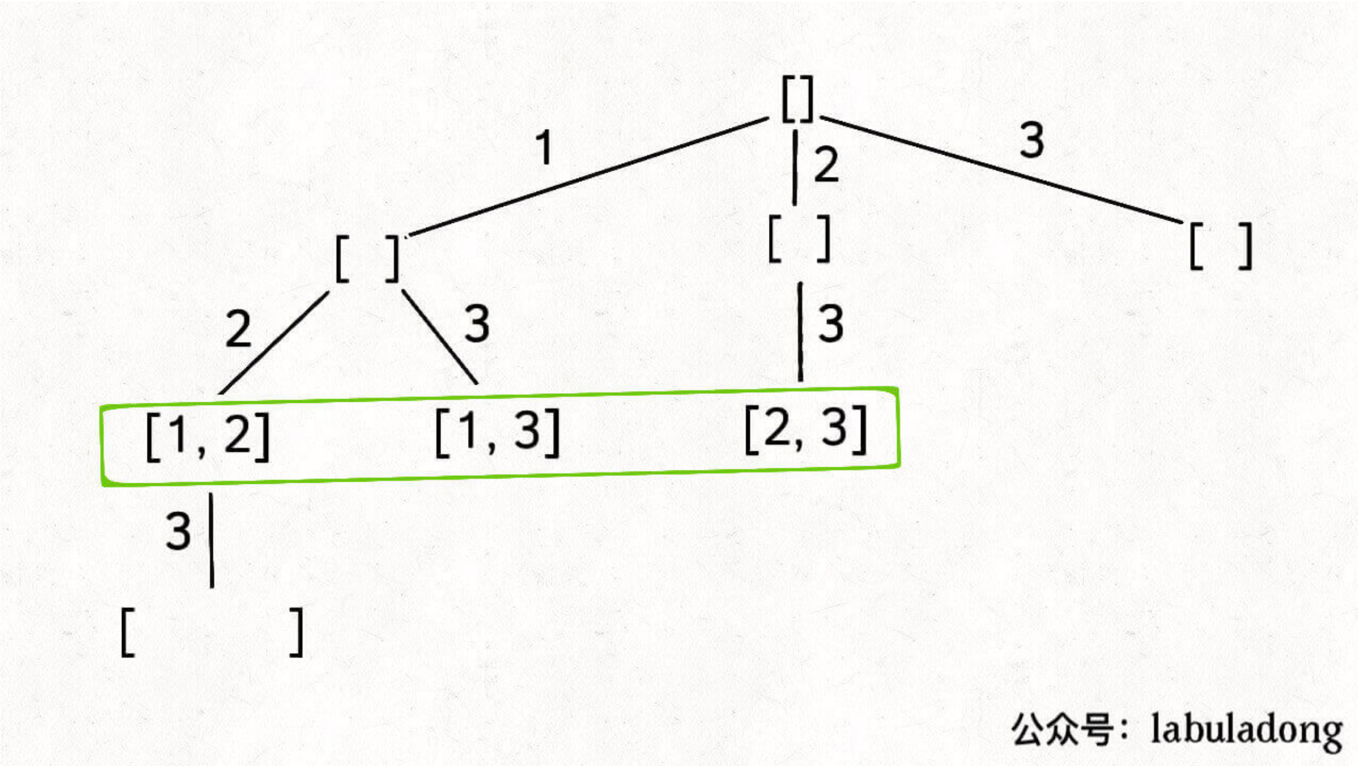
整个推导过程就是这样一棵树：



注意这棵树的特性：

如果把根节点作为第 0 层，将每个节点和根节点之间树枝上的元素作为该节点的值，那么第 `n` 层的所有节点就是大小为 `n` 的所有子集。

你比如大小为 2 的子集就是这一层节点的值：



> **PS：注意，本文之后所说「节点的值」都是指节点和根节点之间树枝上的元素，且将根节点认为是第 0 层。**

那么再进一步，如果想计算所有子集，那只要遍历这棵多叉树，把所有节点的值收集起来不就行了？

直接看代码：

```java
List<List<Integer>> res = new LinkedList<>();
// 记录回溯算法的递归路径
LinkedList<Integer> track = new LinkedList<>();

// 主函数
public List<List<Integer>> subsets(int[] nums) {
    backtrack(nums, 0);
    return res;
}

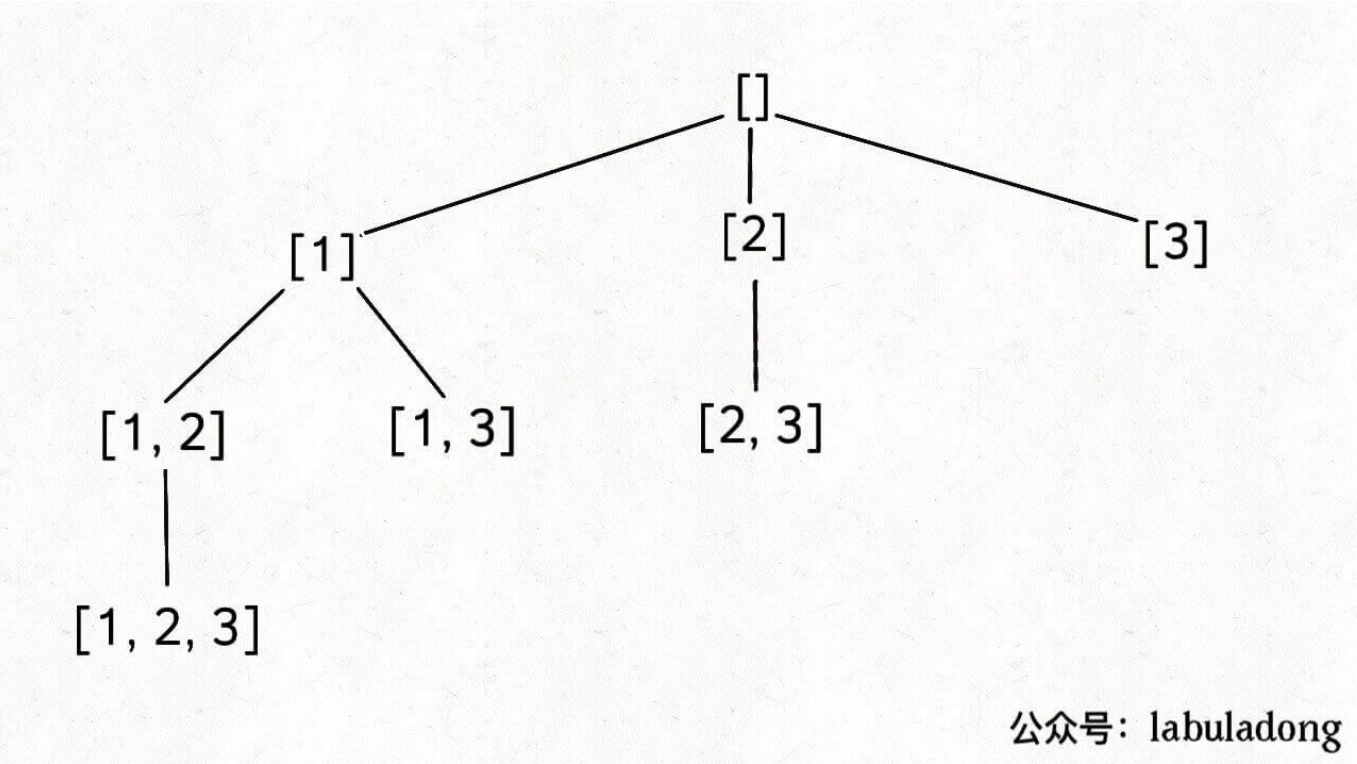
// 回溯算法核心函数，遍历子集问题的回溯树
void backtrack(int[] nums, int start) {

    // 前序位置，每个节点的值都是一个子集
    res.add(new LinkedList<>(track));

    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
```

```
            // 做选择
            track.addLast(nums[i]);
            // 通过 start 参数控制树枝的遍历，避免产生重复的子集
            backtrack(nums, i + 1);
            // 撤销选择
            track.removeLast();
        }
    }
```

看过前文 回溯算法核心框架 的读者应该很容易理解这段代码把，我们使用 `start` 参数控制树枝的生长避免产生重复的子集，用 `track` 记录根节点到每个节点的路径的值，同时在前序位置把每个节点的路径值收集起来，完成回溯树的遍历就收集了所有子集：



最后，`backtrack` 函数开头看似没有 base case，会不会进入无限递归？

其实不会的，当 `start == nums.length` 时，叶子节点的值会被装入 `res`，但 for 循环不会执行，也就结束了递归。

组合（元素无重不可复选）

如果你能够成功的生成所有无重子集，那么你稍微改改代码就能生成所有无重组合了。

你比如说，让你在 `nums = [1,2,3]` 中拿 2 个元素形成所有的组合，你怎么做？

稍微想想就会发现，大小为 2 的所有组合，不就是所有大小为 2 的子集嘛。

所以我说组合和子集是一样的：大小为 `k` 的组合就是大小为 `k` 的子集。

比如力扣第 77 题「组合」：

给定两个整数 `n` 和 `k`，返回范围 `[1, n]` 中所有可能的 `k` 个数的组合。

函数签名如下:

```
List<List<Integer>> combine(int n, int k)
```

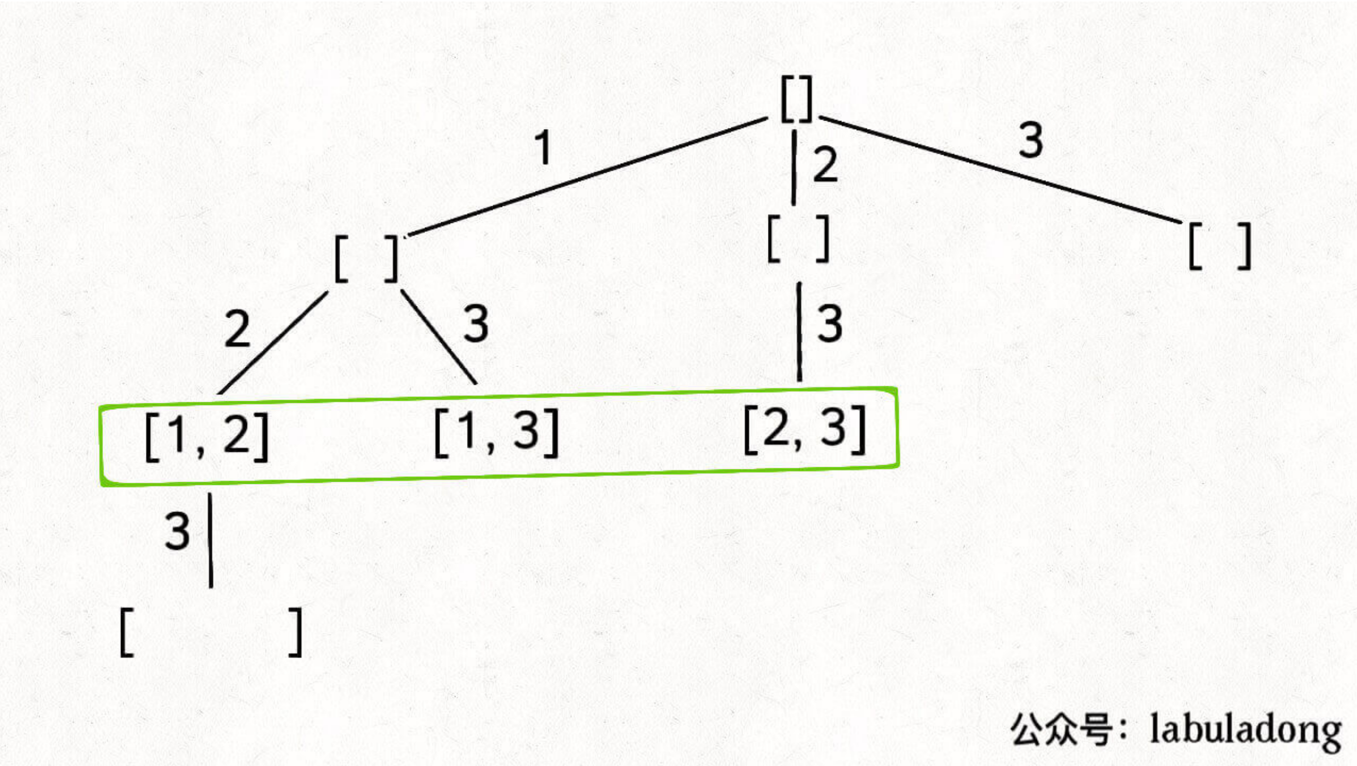
比如 `combine(3, 2)` 的返回值应该是:

```
[ [1,2],[1,3],[2,3] ]
```

这是标准的组合问题，但我给你翻译一下就变成子集问题了:

给你输入一个数组 `nums = [1,2..,n]` 和一个正整数 `k`，请你生成所有大小为 `k` 的子集。

还是以 `nums = [1,2,3]` 为例，刚才让你求所有子集，就是把所有节点的值都收集起来；**现在你只需要把第 2 层（根节点视为第 0 层）的节点收集起来，就是大小为 2 的所有组合**:



反映到代码上，只需要稍改 base case，控制算法仅仅收集第 `k` 层节点的值即可:

```
List<List<Integer>> res = new LinkedList<>();
// 记录回溯算法的递归路径
LinkedList<Integer> track = new LinkedList<>();

// 主函数
public List<List<Integer>> combine(int n, int k) {
    backtrack(1, n, k);
    return res;
}
```

```java
    void backtrack(int start, int n, int k) {
        // base case
        if (k == track.size()) {
            // 遍历到了第 k 层，收集当前节点的值
            res.add(new LinkedList<>(track));
            return;
        }

        // 回溯算法标准框架
        for (int i = start; i <= n; i++) {
            // 选择
            track.addLast(i);
            // 通过 start 参数控制树枝的遍历，避免产生重复的子集
            backtrack(i + 1, n, k);
            // 撤销选择
            track.removeLast();
        }
    }
```

这样，标准的子集问题也解决了。

排列（元素无重不可复选）

排列问题在前文 回溯算法核心框架 讲过，这里就简单过一下。

力扣第 46 题「全排列」就是标准的排列问题：

给定一个**不含重复数字**的数组 `nums`，返回其所有可能的**全排列**。

函数签名如下：

```java
List<List<Integer>> permute(int[] nums)
```

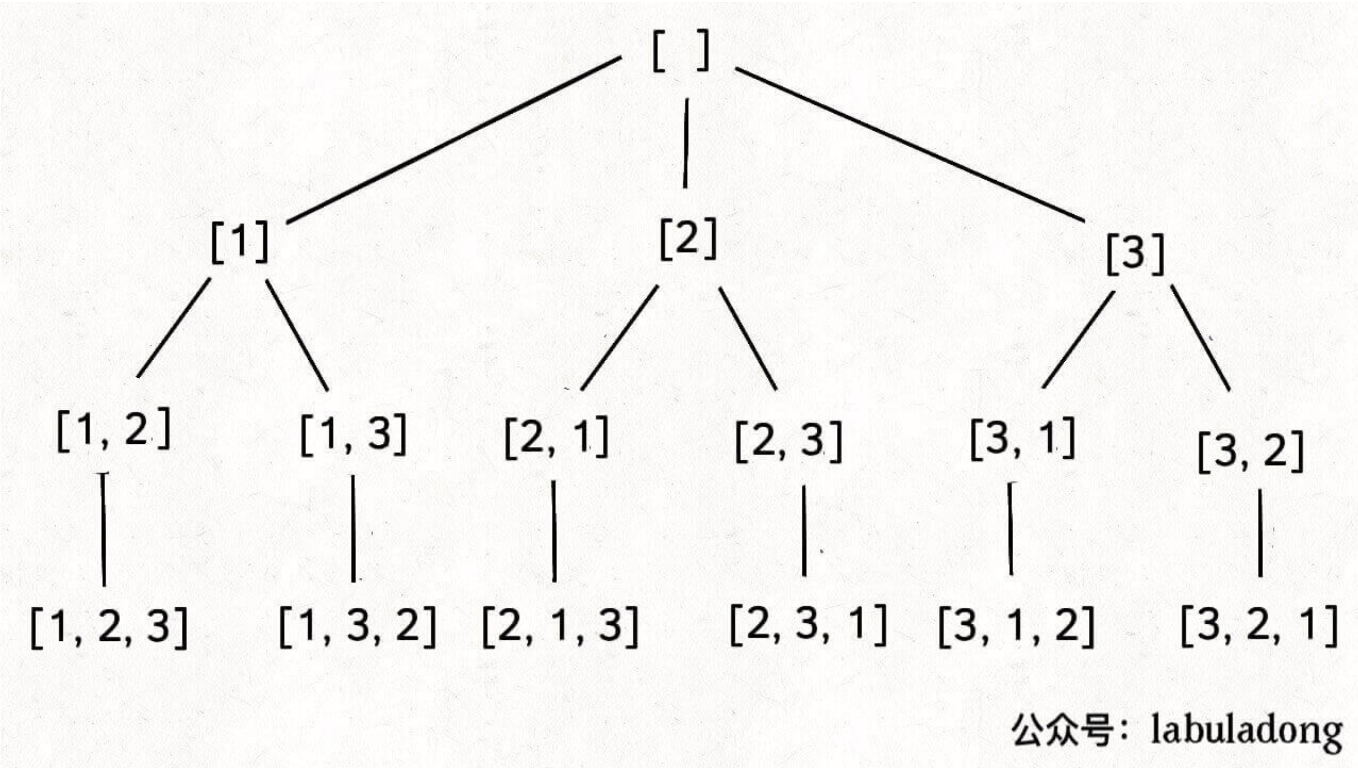
比如输入 `nums = [1,2,3]`，函数的返回值应该是：

```
[
    [1,2,3],[1,3,2],
    [2,1,3],[2,3,1],
    [3,1,2],[3,2,1]
]
```

刚才讲的组合/子集问题使用 `start` 变量保证元素 `nums[start]` 之后只会出现 `nums[start+1..]` 中的元素，通过固定元素的相对位置保证不出现重复的子集。

但排列问题的本质就是穷举元素的位置，`nums[i]` 之后也可以出现 `nums[i]` 左边的元素，所以之前的那一套玩不转了，需要额外使用 `used` 数组来标记哪些元素还可以被选择。

标准全排列可以抽象成如下这棵二叉树：



我们用 used 数组标记已经在路径上的元素避免重复选择，然后收集所有叶子节点上的值，就是所有全排列的结果：

```
List<List<Integer>> res = new LinkedList<>();
// 记录回溯算法的递归路径
LinkedList<Integer> track = new LinkedList<>();
// track 中的元素会被标记为 true
boolean[] used;

/* 主函数，输入一组不重复的数字，返回它们的全排列 */
public List<List<Integer>> permute(int[] nums) {
    used = new boolean[nums.length];
    backtrack(nums);
    return res;
}

// 回溯算法核心函数
void backtrack(int[] nums) {
    // base case，到达叶子节点
    if (track.size() == nums.length) {
        // 收集叶子节点上的值
        res.add(new LinkedList(track));
        return;
    }

    // 回溯算法标准框架
    for (int i = 0; i < nums.length; i++) {
        // 已经存在 track 中的元素，不能重复选择
        if (used[i]) {
            continue;
        }
```

```java
            // 做选择
            used[i] = true;
            track.addLast(nums[i]);
            // 进入下一层回溯树
            backtrack(nums);
            // 取消选择
            track.removeLast();
            used[i] = false;
        }
    }
```

这样，全排列问题就解决了。

但如果题目不让你算全排列，而是让你算元素个数为 `k` 的排列，怎么算？

也很简单，改下 `backtrack` 函数的 base case，仅收集第 `k` 层的节点值即可：

```java
// 回溯算法核心函数
void backtrack(int[] nums, int k) {
    // base case，到达第 k 层
    if (track.size() == k) {
        // 第 k 层节点的值就是大小为 k 的排列
        res.add(new LinkedList(track));
        return;
    }

    // 回溯算法标准框架
    for (int i = 0; i < nums.length; i++) {
        // ...
        backtrack(nums, k);
        // ...
    }
}
```

子集/组合（元素可重不可复选）

刚才讲的标准子集问题输入的 `nums` 是没有重复元素的，但如果存在重复元素，怎么处理呢？

力扣第 90 题「子集 II」就是这样一个问题：

给你一个整数数组 `nums`，其中可能包含重复元素，请你返回该数组所有可能的子集。

函数签名如下：

```java
List<List<Integer>> subsetsWithDup(int[] nums)
```

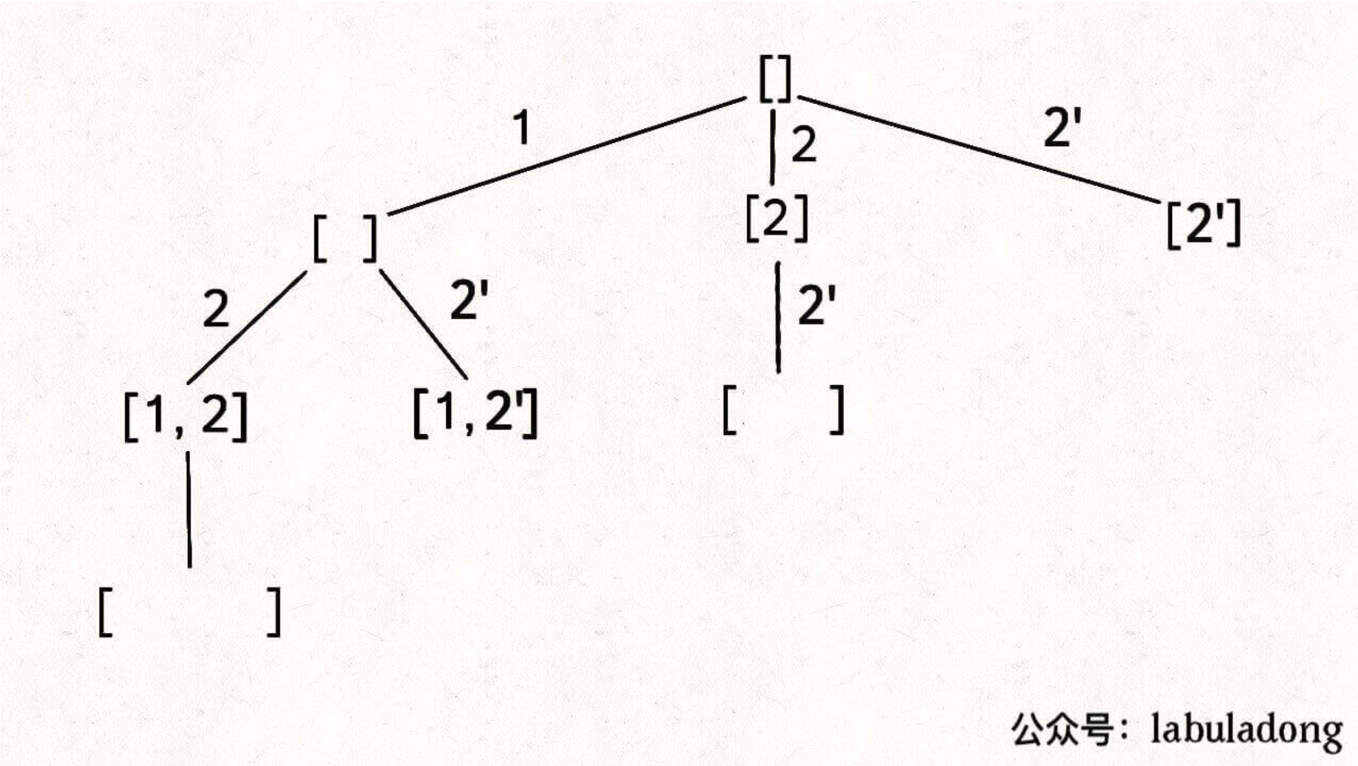
比如输入 `nums = [1,2,2]`，你应该输出：

```
[ [],[1],[2],[1,2],[2,2],[1,2,2] ]
```

当然，按道理说集合不应该包含重复元素的，但既然题目这样问了，我们就忽略这个细节吧，仔细思考一下这道题怎么做才是正事。

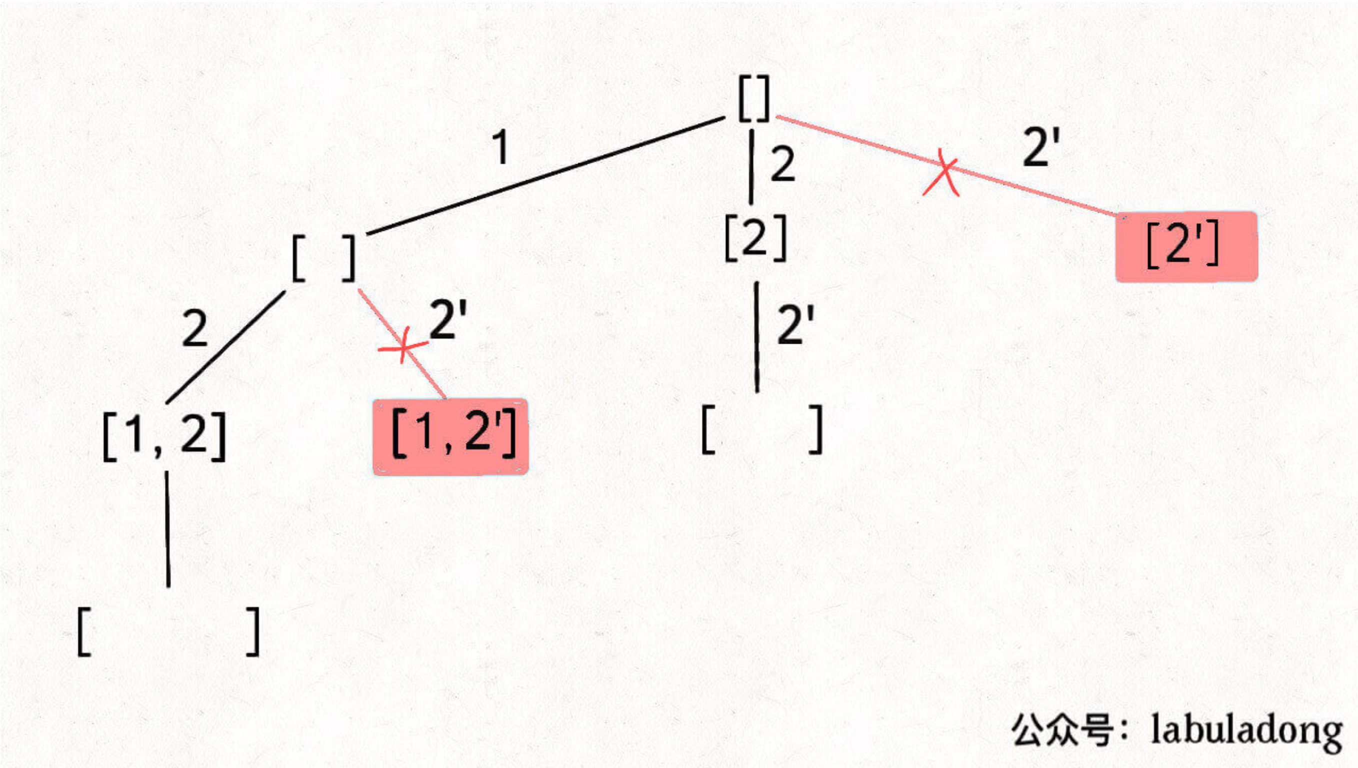
就以 `nums = [1,2,2]` 为例，为了区别两个 `2` 是不同元素，后面我们写作 `nums = [1,2,2']`。

按照之前的思路画出子集的树形结构，显然，两条值相同的相邻树枝会产生重复：




```
[
    [],
    [1],[2],[2'],
    [1,2],[1,2'],[2,2'],
    [1,2,2']
]
```

所以我们需要进行剪枝，如果一个节点有多条值相同的树枝相邻，则只遍历第一条，剩下的都剪掉，不要去遍历：



体现在代码上，需要先进行排序，让相同的元素靠在一起，如果发现 `nums[i] == nums[i-1]`，则跳过：

```
List<List<Integer>> res = new LinkedList<>();
LinkedList<Integer> track = new LinkedList<>();

public List<List<Integer>> subsetsWithDup(int[] nums) {
    // 先排序，让相同的元素靠在一起
    Arrays.sort(nums);
    backtrack(nums, 0);
    return res;
}

void backtrack(int[] nums, int start) {
    // 前序位置，每个节点的值都是一个子集
    res.add(new LinkedList<>(track));

    for (int i = start; i < nums.length; i++) {
        // 剪枝逻辑，值相同的相邻树枝，只遍历第一条
        if (i > start && nums[i] == nums[i - 1]) {
            continue;
        }
        track.addLast(nums[i]);
        backtrack(nums, i + 1);
        track.removeLast();
    }
}
```

这段代码和之前标准的子集问题的代码几乎相同，就是添加了排序和剪枝的逻辑。

至于为什么要这样剪枝，结合前面的图应该也很容易理解，这样带重复元素的子集问题也解决了。

我们说了组合问题和子集问题是等价的，所以我们直接看一道组合的题目吧，这是力扣第 40 题「组合总和 II」：

给你输入 `candidates` 和一个目标和 `target`，从 `candidates` 中找出中所有和为 `target` 的组合。

`candidates` 可能存在重复元素，且其中的每个数字最多只能使用一次。

说这是一个组合问题，其实换个问法就变成子集问题了：请你计算 `candidates` 中所有和为 `target` 的子集。

所以这题怎么做呢？

对比子集问题的解法，只要额外用一个 `trackSum` 变量记录回溯路径上的元素和，然后将 base case 改一改即可解决这道题：

```
List<List<Integer>> res = new LinkedList<>();
// 记录回溯的路径
LinkedList<Integer> track = new LinkedList<>();
// 记录 track 中的元素之和
int trackSum = 0;

public List<List<Integer>> combinationSum2(int[] candidates, int target) {
    if (candidates.length == 0) {
        return res;
    }
    // 先排序，让相同的元素靠在一起
    Arrays.sort(candidates);
    backtrack(candidates, 0, target);
    return res;
}

// 回溯算法主函数
void backtrack(int[] nums, int start, int target) {
    // base case，达到目标和，找到符合条件的组合
    if (trackSum == target) {
        res.add(new LinkedList<>(track));
        return;
    }
    // base case，超过目标和，直接结束
    if (trackSum > target) {
        return;
    }

    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
        // 剪枝逻辑，值相同的树枝，只遍历第一条
        if (i > start && nums[i] == nums[i - 1]) {
            continue;
        }
        // 做选择
        track.add([i]);
        trackSum += nums[i];
        // 递归遍历下一层回溯树
```

```
            backtrack(nums, i + 1, target);
            // 撤销选择
            track.removeLast();
            trackSum -= nums[i];
        }
    }
```

排列（元素可重不可复选）

排列问题的输入如果存在重复，比子集/组合问题稍微复杂一点，我们看看力扣第 47 题「全排列 II」：

给你输入一个可包含重复数字的序列 `nums`，请你写一个算法，返回所有可能的全排列，函数签名如下：

```
List<List<Integer>> permuteUnique(int[] nums)
```

比如输入 `nums = [1,2,2]`，函数返回：

```
[ [1,2,2],[2,1,2],[2,2,1] ]
```

先看解法代码：

```
List<List<Integer>> res = new LinkedList<>();
LinkedList<Integer> track = new LinkedList<>();
boolean[] used;

public List<List<Integer>> permuteUnique(int[] nums) {
    // 先排序，让相同的元素靠在一起
    Arrays.sort(nums);
    used = new boolean[nums.length];
    backtrack(nums, track);
    return res;
}

void backtrack(int[] nums) {
    if (track.size() == nums.length) {
        res.add(new LinkedList(track));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        if (used[i]) {
            continue;
        }
        // 新添加的剪枝逻辑，固定相同的元素在排列中的相对位置
        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
            continue;
        }
```

```
            track.add(nums[i]);
            used[i] = true;
            backtrack(nums);
            track.removeLast();
            used[i] = false;
        }
    }
```

你对比一下之前的标准全排列解法代码，这段解法代码只有两处不同：

1、对 `nums` 进行了排序。

2、添加了一句额外的剪枝逻辑。

类比输入包含重复元素的子集/组合问题，你大概应该理解这么做是为了防止出现重复结果。

但是注意排列问题的剪枝逻辑，和子集/组合问题的剪枝逻辑略有不同：新增了 `!used[i - 1]` 的逻辑判断。

这个地方理解起来就需要一些技巧性了，且听我慢慢到来。为了方便研究，依然把相同的元素用上标 `'` 以示区别。

假设输入为 `nums = [1,2,2']`，标准的全排列算法会得出如下答案：

```
[
    [1,2,2'],[1,2',2],
    [2,1,2'],[2,2',1],
    [2',1,2],[2',2,1]
]
```

显然，这个结果存在重复，比如 `[1,2,2']` 和 `[1,2',2]` 应该只被算作同一个排列，但被算作了两个不同的排列。

所以现在的关键在于，如何设计剪枝逻辑，把这种重复去除掉？

答案是，保证相同元素在排列中的相对位置保持不变。

比如说 `nums = [1,2,2']` 这个例子，我保持排列中 `2` 一直在 `2'` 前面。

这样的话，你从上面 6 个排列中只能挑出 3 个排列符合这个条件：

```
[ [1,2,2'],[2,1,2'],[2,2',1] ]
```

这也就是正确答案。

进一步，如果 `nums = [1,2,2',2'']`，我只要保证重复元素 `2` 的相对位置固定，比如说 `2 -> 2' -> 2''`，也可以得到无重复的全排列结果。

仔细思考，应该很容易明白其中的原理：

标准全排列算法之所以出现重复，是因为把相同元素形成的排列序列视为不同的序列，但实际上它们应该是相同的；而如果固定相同元素形成的序列顺序，当然就避免了重复。

那么反映到代码上，你注意看这个剪枝逻辑：

```
// 新添加的剪枝逻辑，固定相同的元素在排列中的相对位置
if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
    // 如果前面的相邻相等元素没有用过，则跳过
    continue;
}
// 选择 nums[i]
```

当出现重复元素时，比如输入 `nums = [1,2,2',2'']`，`2'` 只有在 `2` 已经被使用的情况下才会被选择，`2''` 只有在 `2'` 已经被使用的情况下才会被选择，这就保证了相同元素在排列中的相对位置保证固定。

好了，这样包含重复输入的排列问题也解决了。

子集/组合（元素无重可复选）

终于到了最后一种类型了：输入数组无重复元素，但每个元素可以被无限次使用。

直接看力扣第 39 题「组合总和」：

给你一个无重复元素的整数数组 `candidates` 和一个目标和 `target`，找出 `candidates` 中可以使数字和为目标数 `target` 的所有组合。`candidates` 中的每个数字可以无限制重复被选取。

函数签名如下：

```
List<List<Integer>> combinationSum(int[] candidates, int target)
```

比如输入 `candidates = [1,2,3], target = 3`，算法应该返回：

```
[ [1,1,1],[1,2],[3] ]
```

这道题说是组合问题，实际上也是子集问题：`candidates` 的哪些子集的和为 `target`？

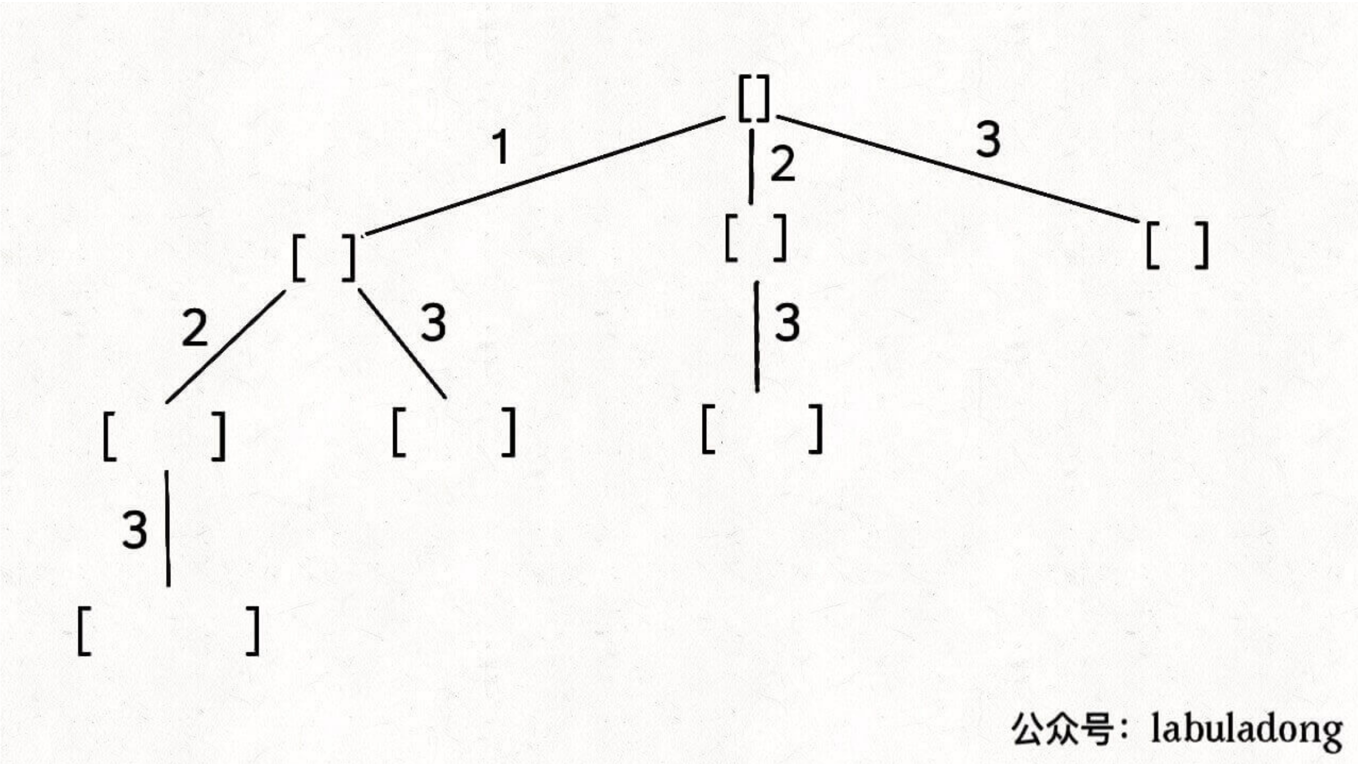
想解决这种类型的问题，也得回到回溯树上，**我们不妨先思考思考，标准的子集/组合问题是如何保证不重复使用元素的？**

答案在于 `backtrack` 递归时输入的参数：

```
// 回溯算法标准框架
for (int i = start; i < nums.length; i++) {
    // ...
    // 递归遍历下一层回溯树，注意参数
    backtrack(nums, i + 1, target);
```

```
    // ...
}
```

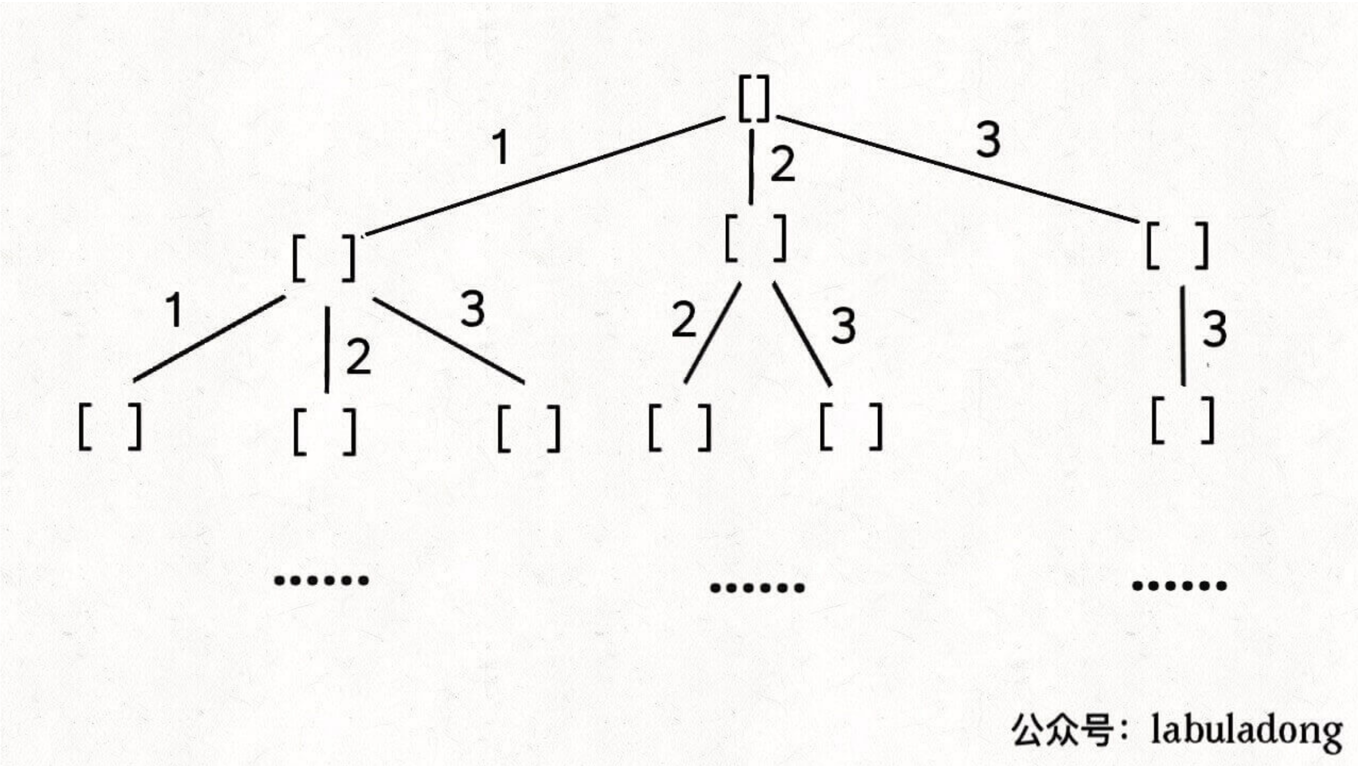
这个 i 从 start 开始，那么下一层回溯树就是从 start + 1 开始，从而保证 nums[start] 这个元素不会被重复使用：



那么反过来，如果我想让每个元素被重复使用，我只要把 i + 1 改成 i 即可：

```
// 回溯算法标准框架
for (int i = start; i < nums.length; i++) {
    // ...
    // 递归遍历下一层回溯树
    backtrack(nums, i, target);
    // ...
}
```

这相当于给之前的回溯树添加了一条树枝，在遍历这棵树的过程中，一个元素可以被无限次使用：



当然，这样这棵回溯树会永远生长下去，所以我们的递归函数需要设置合适的 base case 以结束算法，即路径和大于 `target` 时就没必要再遍历下去了。

这道题的解法代码如下：

```
List<List<Integer>> res = new LinkedList<>();
// 记录回溯的路径
LinkedList<Integer> track = new LinkedList<>();
// 记录 track 中的路径和
int trackSum = 0;

public List<List<Integer>> combinationSum(int[] candidates, int target) {
    if (candidates.length == 0) {
        return res;
    }
    backtrack(candidates, 0, target);
    return res;
}

// 回溯算法主函数
void backtrack(int[] nums, int start, int target) {
    // base case，找到目标和，记录结果
    if (trackSum == target) {
        res.add(new LinkedList<>(track));
        return;
    }
    // base case，超过目标和，停止向下遍历
    if (trackSum > target) {
        return;
    }
```

```
    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
        // 选择 nums[i]
        trackSum += nums[i];
        track.add(nums[i]);
        // 递归遍历下一层回溯树
        // 同一元素可重复使用，注意参数
        backtrack(nums, i, target);
        // 撤销选择 nums[i]
        trackSum -= nums[i];
        track.removeLast();
    }
}
```

排列（元素无重可复选）

力扣上没有类似的题目，我们不妨先想一下，`nums` 数组中的元素无重复且可复选的情况下，会有哪些排列？

比如输入 `nums = [1,2,3]`，那么这种条件下的全排列共有 3^3 = 27 种：

```
[
  [1,1,1],[1,1,2],[1,1,3],[1,2,1],[1,2,2],[1,2,3],[1,3,1],[1,3,2],[1,3,3],
  [2,1,1],[2,1,2],[2,1,3],[2,2,1],[2,2,2],[2,2,3],[2,3,1],[2,3,2],[2,3,3],
  [3,1,1],[3,1,2],[3,1,3],[3,2,1],[3,2,2],[3,2,3],[3,3,1],[3,3,2],[3,3,3]
]
```

标准的全排列算法利用 `used` 数组进行剪枝，避免重复使用同一个元素。如果允许重复使用元素的话，直接放飞自我，去除所有 `used` 数组的剪枝逻辑就行了。

那这个问题就简单了，代码如下：

```
List<List<Integer>> res = new LinkedList<>();
LinkedList<Integer> track = new LinkedList<>();

public List<List<Integer>> permuteRepeat(int[] nums) {
    backtrack(nums);
    return res;
}

// 回溯算法核心函数
void backtrack(int[] nums) {
    // base case，到达叶子节点
    if (track.size() == nums.length) {
        // 收集叶子节点上的值
        res.add(new LinkedList(track));
        return;
    }

    // 回溯算法标准框架
```

```java
    for (int i = 0; i < nums.length; i++) {
        // 做选择
        track.add(nums[i]);
        // 进入下一层回溯树
        backtrack(nums);
        // 取消选择
        track.removeLast();
    }
}
```

至此，排列/组合/子集问题的九种变化就都讲完了。

最后总结

来回顾一下排列/组合/子集问题的三种形式在代码上的区别。

由于子集问题和组合问题本质上是一样的，无非就是 base case 有一些区别，所以把这两个问题放在一起看。

形式一、元素无重不可复选，即 `nums` 中的元素都是唯一的，每个元素最多只能被使用一次，`backtrack` 核心代码如下：

```java
/* 组合/子集问题回溯算法框架 */
void backtrack(int[] nums, int start) {
    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
        // 做选择
        track.addLast(nums[i]);
        // 注意参数
        backtrack(nums, i + 1);
        // 撤销选择
        track.removeLast();
    }
}

/* 排列问题回溯算法框架 */
void backtrack(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        // 剪枝逻辑
        if (used[i]) {
            continue;
        }
        // 做选择
        used[i] = true;
        track.addLast(nums[i]);

        backtrack(nums);
        // 取消选择
        track.removeLast();
        used[i] = false;
```

```
    }
}
```

形式二、元素可重不可复选，即 `nums` 中的元素可以存在重复，每个元素最多只能被使用一次，其关键在于排序和剪枝，`backtrack` 核心代码如下：

```
Arrays.sort(nums);
/* 组合/子集问题回溯算法框架 */
void backtrack(int[] nums, int start) {
    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
        // 剪枝逻辑，跳过值相同的相邻树枝
        if (i > start && nums[i] == nums[i - 1]) {
            continue;
        }
        // 做选择
        track.addLast(nums[i]);
        // 注意参数
        backtrack(nums, i + 1);
        // 撤销选择
        track.removeLast();
    }
}


Arrays.sort(nums);
/* 排列问题回溯算法框架 */
void backtrack(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        // 剪枝逻辑
        if (used[i]) {
            continue;
        }
        // 剪枝逻辑，固定相同的元素在排列中的相对位置
        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
            continue;
        }
        // 做选择
        used[i] = true;
        track.addLast(nums[i]);

        backtrack(nums);
        // 取消选择
        track.removeLast();
        used[i] = false;
    }
}
```

形式三、元素无重可复选，即 `nums` 中的元素都是唯一的，每个元素可以被使用若干次，只要删掉去重逻辑即可，`backtrack` 核心代码如下：

```java
/* 组合/子集问题回溯算法框架 */
void backtrack(int[] nums, int start) {
    // 回溯算法标准框架
    for (int i = start; i < nums.length; i++) {
        // 做选择
        track.addLast(nums[i]);
        // 注意参数
        backtrack(nums, i);
        // 撤销选择
        track.removeLast();
    }
}


/* 排列问题回溯算法框架 */
void backtrack(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        // 做选择
        track.addLast(nums[i]);

        backtrack(nums);
        // 取消选择
        track.removeLast();
    }
}
```

只要从树的角度思考，这些问题看似复杂多变，实则改改 base case 就能解决，这也是为什么我在 学习算法和数据结构的框架思维 和 手把手刷二叉树（纲领篇） 中强调树类型题目重要性的原因。

如果你能够看到这里，真得给你鼓掌，相信你以后遇到各种乱七八糟的算法题，也能一眼看透它们的本质，以不变应万变。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

DFS 算法秒杀所有岛屿题目

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

200. 岛屿数量（中等）

1254. 统计封闭岛屿的数目（中等）

1020. 飞地的数量（中等）

695. 岛屿的最大面积（中等）

1905. 统计子岛屿（中等）

694. 不同的岛屿数量（中等）

岛屿系列算法问题是经典的面试高频题，虽然基本的问题并不难，但是这类问题有一些有意思的扩展，比如求子岛屿数量，求形状不同的岛屿数量等等，本文就来把这些问题一网打尽。

岛屿系列题目的核心考点就是用 DFS/BFS 算法遍历二维数组。

本文主要来讲解如何用 DFS 算法来秒杀岛屿系列题目，不过用 BFS 算法的核心思路是完全一样的，无非就是把 DFS 改写成 BFS 而已。

那么如何在二维矩阵中使用 DFS 搜索呢？如果你把二维矩阵中的每一个位置看做一个节点，这个节点的上下左右四个位置就是相邻节点，那么整个矩阵就可以抽象成一幅网状的「图」结构。

根据 学习数据结构和算法的框架思维，完全可以根据二叉树的遍历框架改写出二维矩阵的 DFS 代码框架：

```
// 二叉树遍历框架
void traverse(TreeNode root) {
    traverse(root.left);
    traverse(root.right);
}

// 二维矩阵遍历框架
void dfs(int[][] grid, int i, int j, boolean[] visited) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
```

```
            // 超出索引边界
            return;
        }
        if (visited[i][j]) {
            // 已遍历过 (i, j)
            return;
        }
        // 进入节点 (i, j)
        visited[i][j] = true;
        dfs(grid, i - 1, j, visited); // 上
        dfs(grid, i + 1, j, visited); // 下
        dfs(grid, i, j - 1, visited); // 左
        dfs(grid, i, j + 1, visited); // 右
    }
```

因为二维矩阵本质上是一幅「图」，所以遍历的过程中需要一个 visited 布尔数组防止走回头路，如果你能理解上面这段代码，那么搞定所有岛屿系列题目都很简单。

这里额外说一个处理二维数组的常用小技巧，你有时会看到使用「方向数组」来处理上下左右的遍历，和前文 图遍历框架 的代码很类似：

```
// 方向数组，分别代表上、下、左、右
int[][] dirs = new int[][]{{-1,0}, {1,0}, {0,-1}, {0,1}};

void dfs(int[][] grid, int i, int j, boolean[] visited) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        // 超出索引边界
        return;
    }
    if (visited[i][j]) {
        // 已遍历过 (i, j)
        return;
    }

    // 进入节点 (i, j)
    visited[i][j] = true;
    // 递归遍历上下左右的节点
    for (int[] d : dirs) {
        int next_i = i + d[0];
        int next_j = j + d[1];
        dfs(grid, next_i, next_j, visited);
    }
    // 离开节点 (i, j)
}
```

这种写法无非就是用 for 循环处理上下左右的遍历罢了，你可以按照个人喜好选择写法。

岛屿数量

这是力扣第 200 题「岛屿数量」，最简单也是最经典的一道问题，题目会输入一个二维数组 grid，其中只包含 0 或者 1，0 代表海水，1 代表陆地，且假设该矩阵四周都是被海水包围着的。

我们说连成片的陆地形成岛屿，那么请你写一个算法，计算这个矩阵 grid 中岛屿的个数，函数签名如下：

```
int numIslands(char[][] grid);
```

比如说题目给你输入下面这个 grid 有四片岛屿，算法应该返回 4：

1	1	0	1	1
1	0	0	0	0
0	0	0	0	1
1	1	0	1	1

思路很简单，关键在于如何寻找并标记「岛屿」，这就要 DFS 算法发挥作用了，我们直接看解法代码：

```
// 主函数，计算岛屿数量
int numIslands(char[][] grid) {
    int res = 0;
    int m = grid.length, n = grid[0].length;
    // 遍历 grid
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == '1') {
                // 每发现一个岛屿，岛屿数量加一
                res++;
                // 然后使用 DFS 将岛屿淹了
                dfs(grid, i, j);
            }
        }
    }
    return res;
}

// 从 (i, j) 开始，将与之相邻的陆地都变成海水
void dfs(char[][] grid, int i, int j) {
    int m = grid.length, n = grid[0].length;
```

```
    if (i < 0 || j < 0 || i >= m || j >= n) {
        // 超出索引边界
        return;
    }
    if (grid[i][j] == '0') {
        // 已经是海水了
        return;
    }
    // 将 (i, j) 变成海水
    grid[i][j] = '0';
    // 淹没上下左右的陆地
    dfs(grid, i + 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i - 1, j);
    dfs(grid, i, j - 1);
}
```

为什么每次遇到岛屿，都要用 DFS 算法把岛屿「淹了」呢？主要是为了省事，避免维护 `visited` 数组。

因为 `dfs` 函数遍历到值为 `0` 的位置会直接返回，所以只要把经过的位置都设置为 `0`，就可以起到不走回头路的作用。

> PS：这类 DFS 算法还有个别名叫做 FloodFill 算法，现在有没有觉得 FloodFill 这个名字还挺贴切的~

这个最最基本的算法问题就说到这，我们来看看后面的题目有什么花样。

封闭岛屿的数量

上一题说二维矩阵四周可以认为也是被海水包围的，所以靠边的陆地也算作岛屿。

力扣第 1254 题「统计封闭岛屿的数目」和上一题有两点不同：

1、用 `0` 表示陆地，用 `1` 表示海水。

2、让你计算「封闭岛屿」的数目。所谓「封闭岛屿」就是上下左右全部被 `1` 包围的 `0`，也就是说**靠边的陆地不算作「封闭岛屿」**。

函数签名如下：

```
int closedIsland(int[][] grid)
```

比如题目给你输入如下这个二维矩阵：

1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	0
1	0	1	0	1	1	1	0
1	0	0	0	0	1	0	1
1	1	1	1	1	1	1	0

算法返回 2，只有图中灰色部分的 `0` 是四周全都被海水包围着的「封闭岛屿」。

那么如何判断「封闭岛屿」呢？其实很简单，把上一题中那些靠边的岛屿排除掉，剩下的不就是「封闭岛屿」了吗？

有了这个思路，就可以直接看代码了，注意这题规定 `0` 表示陆地，用 `1` 表示海水：

```java
// 主函数：计算封闭岛屿的数量
int closedIsland(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    for (int j = 0; j < n; j++) {
        // 把靠上边的岛屿淹掉
        dfs(grid, 0, j);
        // 把靠下边的岛屿淹掉
        dfs(grid, m - 1, j);
    }
    for (int i = 0; i < m; i++) {
        // 把靠左边的岛屿淹掉
        dfs(grid, i, 0);
        // 把靠右边的岛屿淹掉
        dfs(grid, i, n - 1);
    }
    // 遍历 grid，剩下的岛屿都是封闭岛屿
    int res = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 0) {
                res++;
                dfs(grid, i, j);
            }
        }
    }
    return res;
}

// 从 (i, j) 开始，将与之相邻的陆地都变成海水
void dfs(int[][] grid, int i, int j) {
    int m = grid.length, n = grid[0].length;
```

```
    if (i < 0 || j < 0 || i >= m || j >= n) {
        return;
    }
    if (grid[i][j] == 1) {
        // 已经是海水了
        return;
    }
    // 将 (i, j) 变成海水
    grid[i][j] = 1;
    // 淹没上下左右的陆地
    dfs(grid, i + 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i - 1, j);
    dfs(grid, i, j - 1);
}
```

只要提前把靠边的陆地都淹掉，然后算出来的就是封闭岛屿了。

> PS：处理这类岛屿题目除了 DFS/BFS 算法之外，Union Find 并查集算法也是一种可选的方法，前文 Union Find 算法运用 就用 Union Find 算法解决了一道类似的问题。

这道岛屿题目的解法稍微改改就可以解决力扣第 1020 题「飞地的数量」，这题不让你求封闭岛屿的数量，而是求封闭岛屿的面积总和。

其实思路都是一样的，先把靠边的陆地淹掉，然后去数剩下的陆地数量就行了，注意第 1020 题中 `1` 代表陆地，`0` 代表海水：

```
int numEnclaves(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    // 淹掉靠边的陆地
    for (int i = 0; i < m; i++) {
        dfs(grid, i, 0);
        dfs(grid, i, n - 1);
    }
    for (int j = 0; j < n; j++) {
        dfs(grid, 0, j);
        dfs(grid, m - 1, j);
    }

    // 数一数剩下的陆地
    int res = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                res += 1;
            }
        }
    }

    return res;
}
```

```
// 和之前的实现类似
void dfs(int[][] grid, int i, int j) {
    // ...
}
```

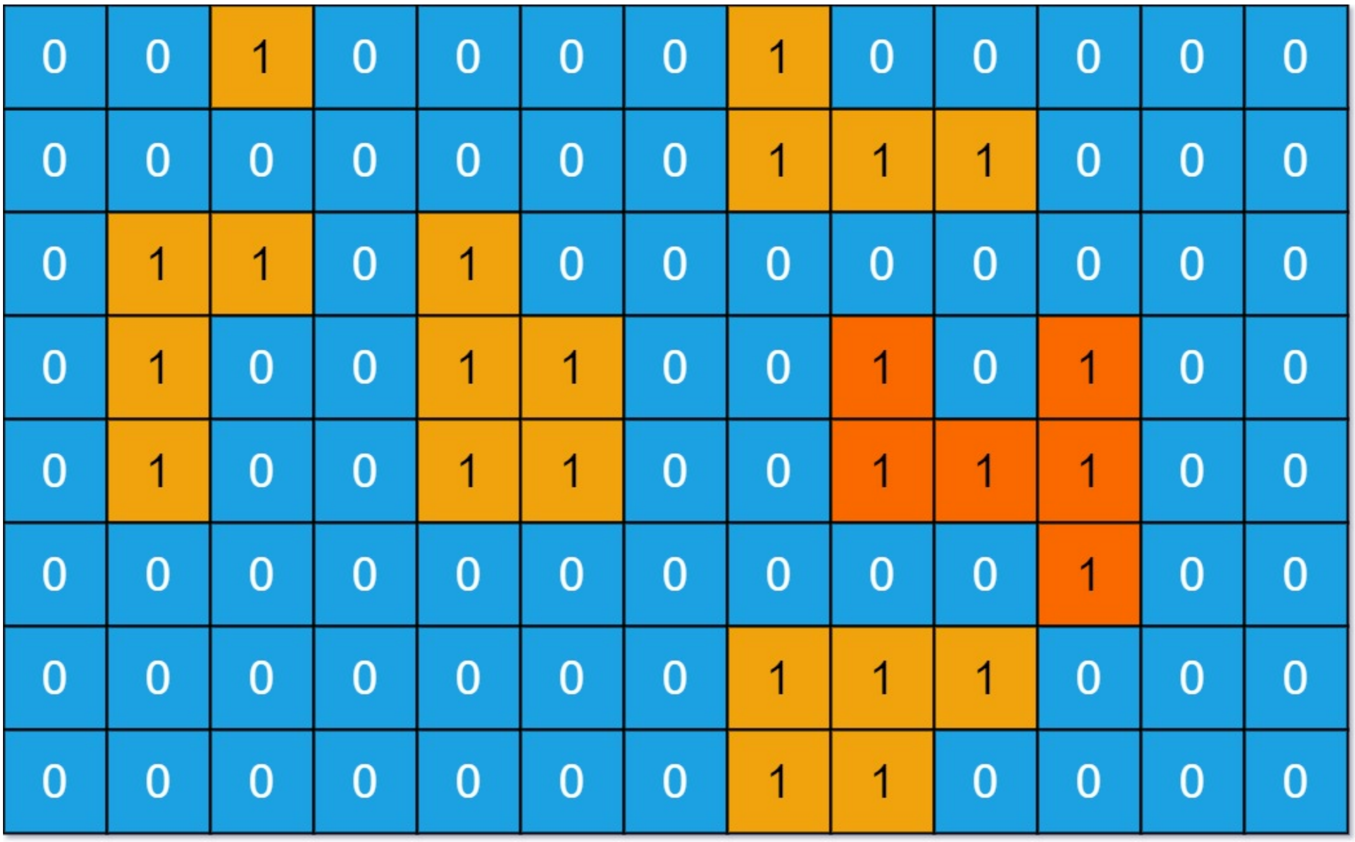
篇幅所限，具体代码我就不写了，我们继续看其他的岛屿题目。

岛屿的最大面积

这是力扣第 695 题「岛屿的最大面积」，`0` 表示海水，`1` 表示陆地，现在不让你计算岛屿的个数了，而是让你计算最大的那个岛屿的面积，函数签名如下：

```
int maxAreaOfIsland(int[][] grid)
```

比如题目给你输入如下一个二维矩阵：



其中面积最大的是橘红色的岛屿，算法返回它的面积 6。

这题的大体思路和之前完全一样，只不过 `dfs` 函数淹没岛屿的同时，还应该想办法记录这个岛屿的面积。

我们可以给 `dfs` 函数设置返回值，记录每次淹没的陆地的个数，直接看解法吧：

```
int maxAreaOfIsland(int[][] grid) {
    // 记录岛屿的最大面积
    int res = 0;
```

```java
    int m = grid.length, n = grid[0].length;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                // 淹没岛屿，并更新最大岛屿面积
                res = Math.max(res, dfs(grid, i, j));
            }
        }
    }
    return res;
}

// 淹没与 (i, j) 相邻的陆地，并返回淹没的陆地面积
int dfs(int[][] grid, int i, int j) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        // 超出索引边界
        return 0;
    }
    if (grid[i][j] == 0) {
        // 已经是海水了
        return 0;
    }
    // 将 (i, j) 变成海水
    grid[i][j] = 0;

    return dfs(grid, i + 1, j)
         + dfs(grid, i, j + 1)
         + dfs(grid, i - 1, j)
         + dfs(grid, i, j - 1) + 1;
}
```

解法和之前相比差不多，我也不多说了，接下来的两道岛屿题目是比较有技巧性的，我们重点来看一下。

子岛屿数量

如果说前面的题目都是模板题，那么力扣第 1905 题「统计子岛屿」可能得动动脑子了：

1905. 统计子岛屿

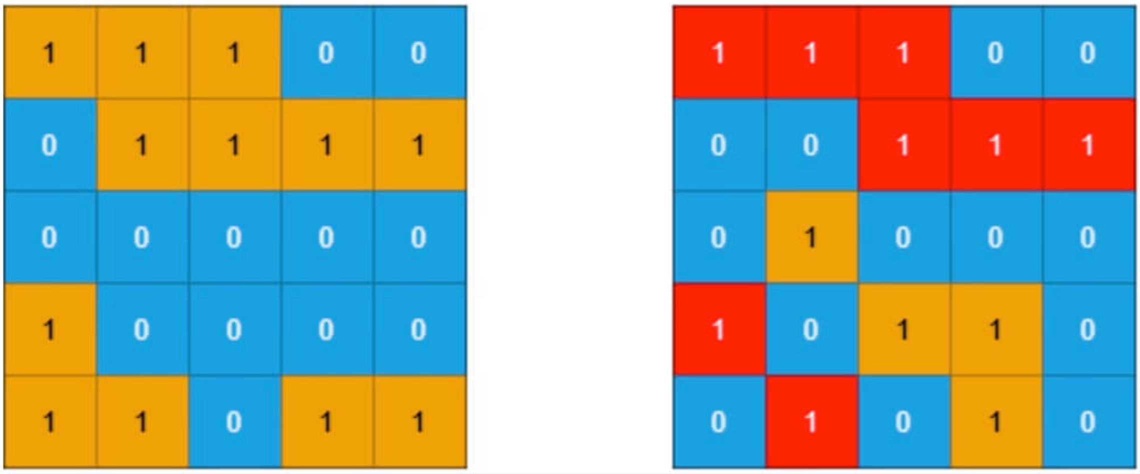
难度 中等 22 收藏 分享 切换为英文 接收动态 反馈

给你两个 `m x n` 的二进制矩阵 `grid1` 和 `grid2` ，它们只包含 `0` （表示水域）和 `1` （表示陆地）。一个 **岛屿** 是由 **四个方向** （水平或者竖直）上相邻的 `1` 组成的区域。任何矩阵以外的区域都视为水域。

如果 `grid2` 的一个岛屿，被 `grid1` 的一个岛屿 **完全** 包含，也就是说 `grid2` 中该岛屿的每一个格子都被 `grid1` 中 **同一个** 岛屿完全包含，那么我们称 `grid2` 中的这个岛屿为 **子岛屿** 。

请你返回 `grid2` 中 **子岛屿** 的数目 。

示例 1：



输入： grid1 = [[1,1,1,0,0],[0,1,1,1,1],[0,0,0,0,0],[1,0,0,0,0],[1,1,0,1,1]], grid2 = [[1,1,1,0,0],[0,0,1,1,1],[0,1,0,0,0],[1,0,1,1,0],[0,1,0,1,0]]
输出： 3
解释： 如上图所示，左边为 grid1 ，右边为 grid2 。
grid2 中标红的 1 区域是子岛屿，总共有 3 个子岛屿。

这道题的关键在于，如何快速判断子岛屿？肯定可以借助 Union Find 并查集算法 来判断，不过本文重点在 DFS 算法，就不展开并查集算法了。

什么情况下 `grid2` 中的一个岛屿 `B` 是 `grid1` 中的一个岛屿 `A` 的子岛？

当岛屿 `B` 中所有陆地在岛屿 `A` 中也是陆地的时候，岛屿 `B` 是岛屿 `A` 的子岛。

反过来说，如果岛屿 `B` 中存在一片陆地，在岛屿 `A` 的对应位置是海水，那么岛屿 `B` 就不是岛屿 `A` 的子岛。

那么，我们只要遍历 `grid2` 中的所有岛屿，把那些不可能是子岛的岛屿排除掉，剩下的就是子岛。

依据这个思路，可以直接写出下面的代码：

```
int countSubIslands(int[][] grid1, int[][] grid2) {
    int m = grid1.length, n = grid1[0].length;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid1[i][j] == 0 && grid2[i][j] == 1) {
                // 这个岛屿肯定不是子岛，淹掉
                dfs(grid2, i, j);
            }
        }
    }
    // 现在 grid2 中剩下的岛屿都是子岛，计算岛屿数量
    int res = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid2[i][j] == 1) {
                res++;
                dfs(grid2, i, j);
            }
        }
    }
    return res;
}

// 从 (i, j) 开始，将与之相邻的陆地都变成海水
void dfs(int[][] grid, int i, int j) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        return;
    }
    if (grid[i][j] == 0) {
        return;
    }

    grid[i][j] = 0;
    dfs(grid, i + 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i - 1, j);
    dfs(grid, i, j - 1);
}
```

这道题的思路和计算「封闭岛屿」数量的思路有些类似，只不过后者排除那些靠边的岛屿，前者排除那些不可能是子岛的岛屿。

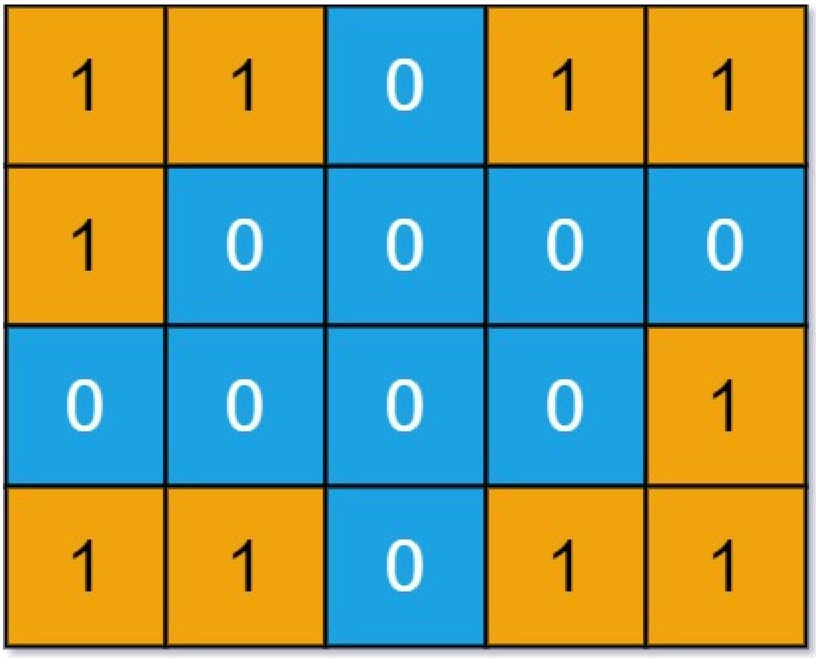
不同的岛屿数量

这是本文的最后一道岛屿题目，作为压轴题，当然是最有意思的。

力扣第 694 题「不同的岛屿数量」，题目还是输入一个二维矩阵，`0` 表示海水，`1` 表示陆地，这次让你计算 **不同的 (distinct)** 岛屿数量，函数签名如下：

```
int numDistinctIslands(int[][] grid)
```

比如题目输入下面这个二维矩阵：



其中有四个岛屿，但是左下角和右上角的岛屿形状相同，所以不同的岛屿共有三个，算法返回 3。

很显然我们得想办法把二维矩阵中的「岛屿」进行转化，变成比如字符串这样的类型，然后利用 HashSet 这样的数据结构去重，最终得到不同的岛屿的个数。

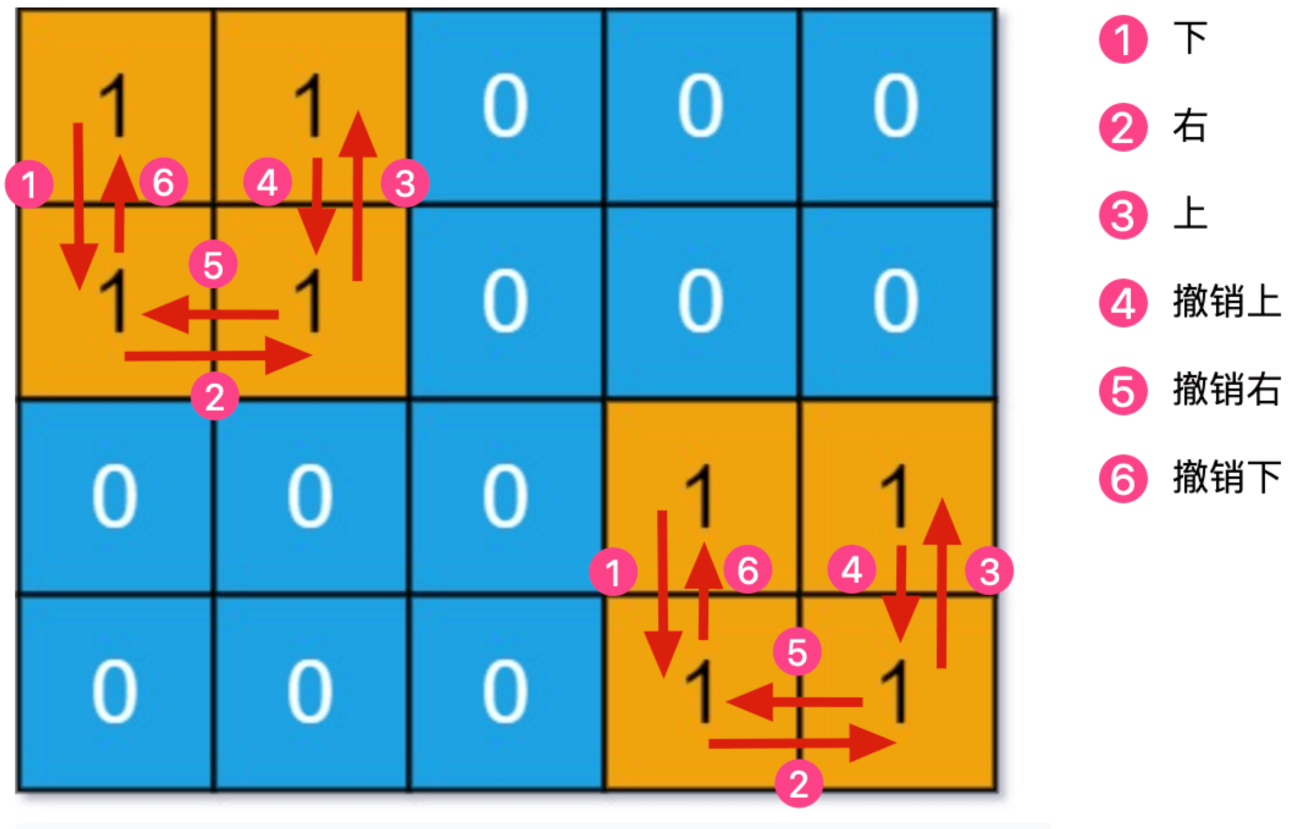
如果想把岛屿转化成字符串，说白了就是序列化，序列化说白了就是遍历嘛，前文 二叉树的序列化和反序列化 讲了二叉树和字符串互转，这里也是类似的。

首先，对于形状相同的岛屿，如果从同一起点出发，`dfs` 函数遍历的顺序肯定是一样的。

因为遍历顺序是写死在你的递归函数里面的，不会动态改变：

```
void dfs(int[][] grid, int i, int j) {
    // 递归顺序:
    dfs(grid, i - 1, j); // 上
    dfs(grid, i + 1, j); // 下
    dfs(grid, i, j - 1); // 左
    dfs(grid, i, j + 1); // 右
}
```

所以，遍历顺序从某种意义上说就可以用来描述岛屿的形状，比如下图这两个岛屿：



假设它们的遍历顺序是：

> 下，右，上，撤销上，撤销右，撤销下

如果我用分别用 `1, 2, 3, 4` 代表上下左右，用 `-1, -2, -3, -4` 代表上下左右的撤销，那么可以这样表示它们的遍历顺序：

> 2, 4, 1, -1, -4, -2

你看，这就相当于是岛屿序列化的结果，只要每次使用 `dfs` 遍历岛屿的时候生成这串数字进行比较，就可以计算到底有多少个不同的岛屿了。

我们需要稍微改造 `dfs` 函数，添加一些函数参数以便记录遍历顺序：

```
void dfs(int[][] grid, int i, int j, StringBuilder sb, int dir) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n
        || grid[i][j] == 0) {
        return;
    }
    // 前序遍历位置：进入 (i, j)
    grid[i][j] = 0;
    sb.append(dir).append(',');

    dfs(grid, i - 1, j, sb, 1); // 上
    dfs(grid, i + 1, j, sb, 2); // 下
    dfs(grid, i, j - 1, sb, 3); // 左
    dfs(grid, i, j + 1, sb, 4); // 右
```

```java
        // 后序遍历位置：离开 (i, j)
        sb.append(-dir).append(',');
    }
```

dir 记录方向，dfs 函数递归结束后，sb 记录着整个遍历顺序，其实这就是前文 回溯算法核心套路 说到的回溯算法框架，你看到头来这些算法都是相通的。

有了这个 dfs 函数就好办了，我们可以直接写出最后的解法代码：

```java
int numDistinctIslands(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    // 记录所有岛屿的序列化结果
    HashSet<String> islands = new HashSet<>();
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                // 淹掉这个岛屿，同时存储岛屿的序列化结果
                StringBuilder sb = new StringBuilder();
                // 初始的方向可以随便写，不影响正确性
                dfs(grid, i, j, sb, 666);
                islands.add(sb.toString());
            }
        }
    }
    // 不相同的岛屿数量
    return islands.size();
}
```

这样，这道题就解决了，至于为什么初始调用 dfs 函数时的 dir 参数可以随意写，这里涉及 DFS 和回溯算法的一个细微差别，前文 图算法基础 有写，这里就不展开了。

以上就是全部岛屿系列题目的解题思路，也许前面的题目大部分人会做，但是最后两题还是比较巧妙的，希望本文对你有帮助。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜

labuladong公众号

3.2 BFS 算法

BFS 算法起源于二叉树的层序遍历，其核心是利用队列这种数据结构。

且 BFS 算法常见于求最值的场景，因为 BFS 的算法逻辑保证了算法第一次到达目标时的代价是最小的。

BFS 算法解题套路框架

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

111. 二叉树的最小深度（简单）

752. 打开转盘锁（中等）

后台有很多人问起 BFS 和 DFS 的框架，今天就来说说吧。

首先，你要说我没写过 BFS 框架，这话没错，今天写个框架你背住就完事儿了。但要是说没写过 DFS 框架，那你还真是说错了，**其实 DFS 算法就是回溯算法**，我们前文 回溯算法框架套路详解 就写过了，而且写得不是一般得好，建议好好复习，嘿嘿嘿~

BFS 的核心思想应该不难理解的，就是把一些问题抽象成图，从一个点开始，向四周开始扩散。一般来说，我们写 BFS 算法都是用「队列」这种数据结构，每次将一个节点周围的所有节点加入队列。

BFS 相对 DFS 的最主要的区别是：**BFS 找到的路径一定是最短的，但代价就是空间复杂度可能比 DFS 大很多**，至于为什么，我们后面介绍了框架就很容易看出来了。

本文就由浅入深写两道 BFS 的典型题目，分别是「二叉树的最小高度」和「打开密码锁的最少步数」，手把手教你怎么写 BFS 算法。

一、算法框架

要说框架的话，我们先举例一下 BFS 出现的常见场景好吧，**问题的本质就是让你在一幅「图」中找到从起点 `start` 到终点 `target` 的最近距离，这个例子听起来很枯燥，但是 BFS 算法问题其实都是在干这个事儿**，把枯燥的本质搞清楚了，再去欣赏各种问题的包装才能胸有成竹嘛。

这个广义的描述可以有各种变体，比如走迷宫，有的格子是围墙不能走，从起点到终点的最短距离是多少？如果这个迷宫带「传送门」可以瞬间传送呢？

再比如说两个单词，要求你通过某些替换，把其中一个变成另一个，每次只能替换一个字符，最少要替换几次？

再比如说连连看游戏，两个方块消除的条件不仅仅是图案相同，还得保证两个方块之间的最短连线不能多于两个拐点。你玩连连看，点击两个坐标，游戏是如何判断它俩的最短连线有几个拐点的？

再比如……

净整些花里胡哨的，这些问题都没啥奇技淫巧，本质上就是一幅「图」，让你从一个起点，走到终点，问最短路径。这就是 BFS 的本质，框架搞清楚了直接默写就好。



记住下面这个框架就 OK 了：

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路

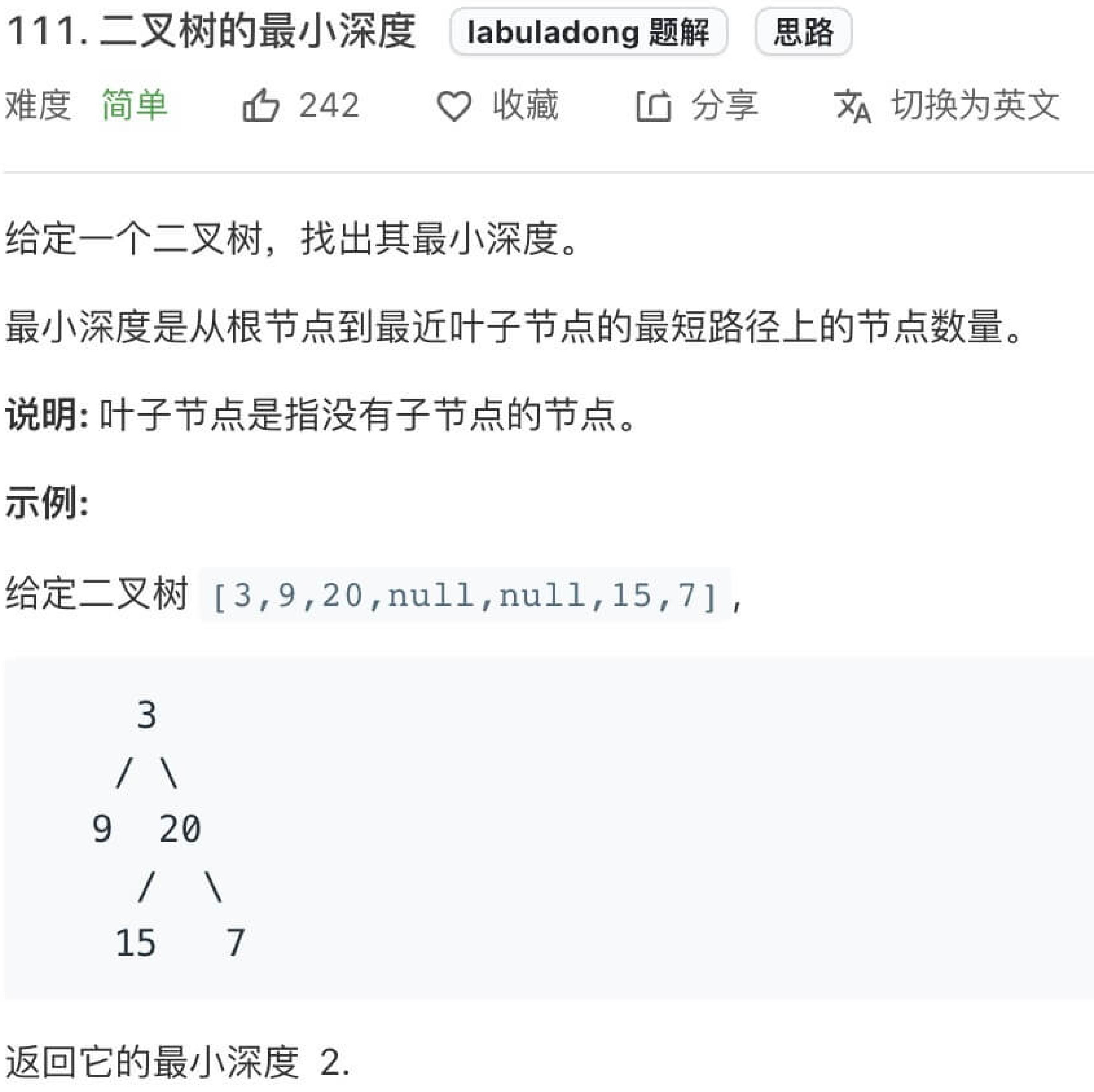
    q.offer(start); // 将起点加入队列
    visited.add(start);
    int step = 0; // 记录扩散的步数

    while (q not empty) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            /* 划重点：这里判断是否到达终点 */
            if (cur is target)
                return step;
            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj()) {
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
            }
        }
        /* 划重点：更新步数在这里 */
        step++;
    }
}
```

队列 `q` 就不说了，BFS 的核心数据结构；`cur.adj()` 泛指 `cur` 相邻的节点，比如说二维数组中，`cur` 上下左右四面的位置就是相邻节点；`visited` 的主要作用是防止走回头路，大部分时候都是必须的，但是像一般的二叉树结构，没有子节点到父节点的指针，不会走回头路就不需要 `visited`。

二、二叉树的最小高度

先来个简单的问题实践一下 BFS 框架吧，判断一棵二叉树的**最小**高度，这也是力扣第 111 题「[二叉树的最小深度](#)」：



111. 二叉树的最小深度 labuladong 题解 思路

难度 简单 242 收藏 分享 切换为英文

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明: 叶子节点是指没有子节点的节点。

示例:

给定二叉树 `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最小深度 2.

怎么套到 BFS 的框架里呢？首先明确一下起点 `start` 和终点 `target` 是什么，怎么判断到达了终点？

显然起点就是 `root` 根节点，终点就是最靠近根节点的那个「叶子节点」嘛，叶子节点就是两个子节点都是 `null` 的节点：

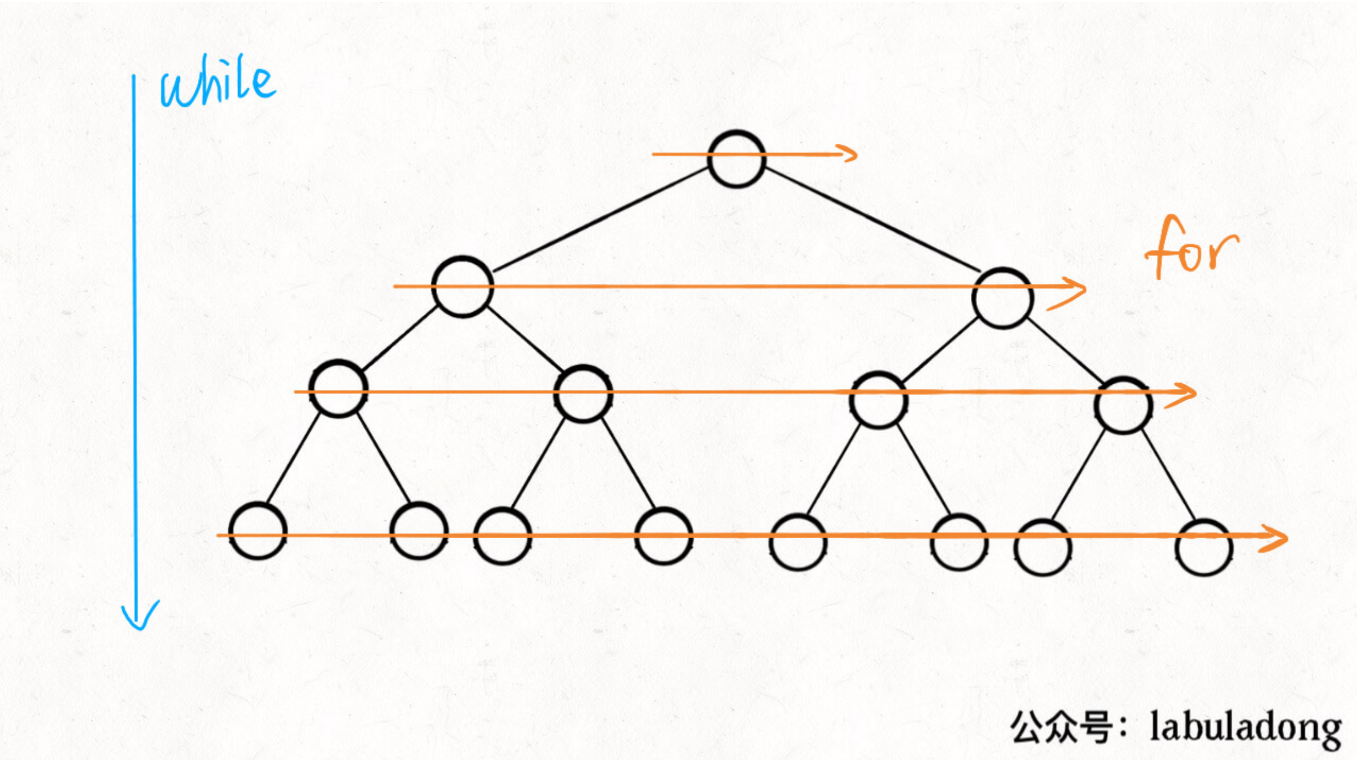
```
if (cur.left == null && cur.right == null)
    // 到达叶子节点
```

那么，按照我们上述的框架稍加改造来写解法即可：

```java
int minDepth(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);
    // root 本身就是一层，depth 初始化为 1
    int depth = 1;

    while (!q.isEmpty()) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            /* 判断是否到达终点 */
            if (cur.left == null && cur.right == null)
                return depth;
            /* 将 cur 的相邻节点加入队列 */
            if (cur.left != null)
                q.offer(cur.left);
            if (cur.right != null)
                q.offer(cur.right);
        }
        /* 这里增加步数 */
        depth++;
    }
    return depth;
}
```

这里注意这个 `while` 循环和 `for` 循环的配合，**`while` 循环控制一层一层往下走，`for` 循环利用 `sz` 变量控制从左到右遍历每一层二叉树节点**：



这一点很重要，这个形式在普通 BFS 问题中都很常见，但是在 Dijkstra 算法模板框架 中我们修改了这种代码模式，读完并理解本文后你可以去看看 BFS 算法是如何演变成 Dijkstra 算法在加权图中寻找最短路径的。

话说回来，二叉树本身是很简单的数据结构，我想上述代码你应该可以理解的，其实其他复杂问题都是这个框架的变形，再探讨复杂问题之前，我们解答两个问题：

1、为什么 BFS 可以找到最短距离，DFS 不行吗？

首先，你看 BFS 的逻辑，`depth` 每增加一次，队列中的所有节点都向前迈一步，这保证了第一次到达终点的时候，走的步数是最少的。

DFS 不能找最短路径吗？其实也是可以的，但是时间复杂度相对高很多。你想啊，DFS 实际上是靠递归的堆栈记录走过的路径，你要找到最短路径，肯定得把二叉树中所有树杈都探索完才能对比出最短的路径有多长对不对？而 BFS 借助队列做到一次一步「齐头并进」，是可以在不遍历完整棵树的条件下找到最短距离的。

形象点说，DFS 是线，BFS 是面；DFS 是单打独斗，BFS 是集体行动。这个应该比较容易理解吧。

2、既然 BFS 那么好，为啥 DFS 还要存在？

BFS 可以找到最短距离，但是空间复杂度高，而 DFS 的空间复杂度较低。

还是拿刚才我们处理二叉树问题的例子，假设给你的这个二叉树是满二叉树，节点数为 `N`，对于 DFS 算法来说，空间复杂度无非就是递归堆栈，最坏情况下顶多就是树的高度，也就是 `O(logN)`。

但是你想想 BFS 算法，队列中每次都会储存着二叉树一层的节点，这样的话最坏情况下空间复杂度应该是树的最底层节点的数量，也就是 `N/2`，用 Big O 表示的话也就是 `O(N)`。

由此观之，BFS 还是有代价的，一般来说在找最短路径的时候使用 BFS，其他时候还是 DFS 使用得多一些（主要是递归代码好写）。

好了，现在你对 BFS 了解得足够多了，下面来一道难一点的题目，深化一下框架的理解吧。

三、解开密码锁的最少次数

这道 LeetCode 题目是第 752 题，比较有意思：

752. 打开转盘锁 labuladong 题解 思路

难度 中等 97 收藏 分享 切换为英文 关注 反馈

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字： '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' 。每个拨轮可以自由旋转：例如把 '9' 变为 '0'，'0' 变为 '9' 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 '0000' ，一个代表四个拨轮的数字的字符串。

列表 deadends 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 target 代表可以解锁的数字，你需要给出最小的旋转次数，如果无论如何不能解锁，返回 -1。

示例 1:

```
输入：deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出：6
解释：
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的，
因为当拨动到 "0102" 时这个锁就会被锁定。
```

示例 2:

```
输入：deadends = ["8888"], target = "0009"
输出：1
解释：
把最后一位反向旋转一次即可 "0000" -> "0009"。
```

示例 3:

```
输入：deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], target = "8888"
输出：-1
解释：
无法旋转到目标数字且不被锁定。
```

题目中描述的就是我们生活中常见的那种密码锁，若果没有任何约束，最少的拨动次数很好算，就像我们平时开密码锁那样直奔密码拨就行了。

但现在的难点就在于，不能出现 deadends，应该如何计算出最少的转动次数呢？

第一步，我们不管所有的限制条件，不管 deadends 和 target 的限制，就思考一个问题：如果让你设计一个算法，穷举所有可能的密码组合，你怎么做？

穷举呗，再简单一点，如果你只转一下锁，有几种可能？总共有 4 个位置，每个位置可以向上转，也可以向下转，也就是有 8 种可能对吧。

比如说从 "0000" 开始，转一次，可以穷举出 "1000", "9000", "0100", "0900"... 共 8 种密码。然后，再以这 8 种密码作为基础，对每个密码再转一下，穷举出所有可能...

仔细想想，这就可以抽象成一幅图，每个节点有 8 个相邻的节点，又让你求最短距离，这不就是典型的 BFS 嘛，框架就可以派上用场了，先写出一个「简陋」的 BFS 框架代码再说别的：

```
// 将 s[j] 向上拨动一次
String plusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '9')
        ch[j] = '0';
    else
        ch[j] += 1;
    return new String(ch);
}
// 将 s[i] 向下拨动一次
String minusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '0')
        ch[j] = '9';
    else
        ch[j] -= 1;
    return new String(ch);
}

// BFS 框架，打印出所有可能的密码
void BFS(String target) {
    Queue<String> q = new LinkedList<>();
    q.offer("0000");

    while (!q.isEmpty()) {
        int sz = q.size();
        /* 将当前队列中的所有节点向周围扩散 */
        for (int i = 0; i < sz; i++) {
            String cur = q.poll();
            /* 判断是否到达终点 */
            System.out.println(cur);

            /* 将一个节点的相邻节点加入队列 */
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                String down = minusOne(cur, j);
                q.offer(up);
                q.offer(down);
            }
        }
        /* 在这里增加步数 */
    }
```

```
        return;
    }
```

> PS：这段代码当然有很多问题，但是我们做算法题肯定不是一蹴而就的，而是从简陋到完美的。不要完美主义，咱要慢慢来，好不。

这段 BFS 代码已经能够穷举所有可能的密码组合了，但是显然不能完成题目，有如下问题需要解决：

1、会走回头路。比如说我们从 `"0000"` 拨到 `"1000"`，但是等从队列拿出 `"1000"` 时，还会拨出一个 `"0000"`，这样的话会产生死循环。

2、没有终止条件，按照题目要求，我们找到 `target` 就应该结束并返回拨动的次数。

3、没有对 `deadends` 的处理，按道理这些「死亡密码」是不能出现的，也就是说你遇到这些密码的时候需要跳过。

如果你能够看懂上面那段代码，真得给你鼓掌，只要按照 BFS 框架在对应的位置稍作修改即可修复这些问题：

```java
int openLock(String[] deadends, String target) {
    // 记录需要跳过的死亡密码
    Set<String> deads = new HashSet<>();
    for (String s : deadends) deads.add(s);
    // 记录已经穷举过的密码，防止走回头路
    Set<String> visited = new HashSet<>();
    Queue<String> q = new LinkedList<>();
    // 从起点开始启动广度优先搜索
    int step = 0;
    q.offer("0000");
    visited.add("0000");

    while (!q.isEmpty()) {
        int sz = q.size();
        /* 将当前队列中的所有节点向周围扩散 */
        for (int i = 0; i < sz; i++) {
            String cur = q.poll();

            /* 判断是否到达终点 */
            if (deads.contains(cur))
                continue;
            if (cur.equals(target))
                return step;

            /* 将一个节点的未遍历相邻节点加入队列 */
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                if (!visited.contains(up)) {
                    q.offer(up);
                    visited.add(up);
                }
                String down = minusOne(cur, j);
```

```
                if (!visited.contains(down)) {
                    q.offer(down);
                    visited.add(down);
                }
            }
        }
        /* 在这里增加步数 */
        step++;
    }
    // 如果穷举完都没找到目标密码，那就是找不到了
    return -1;
}
```

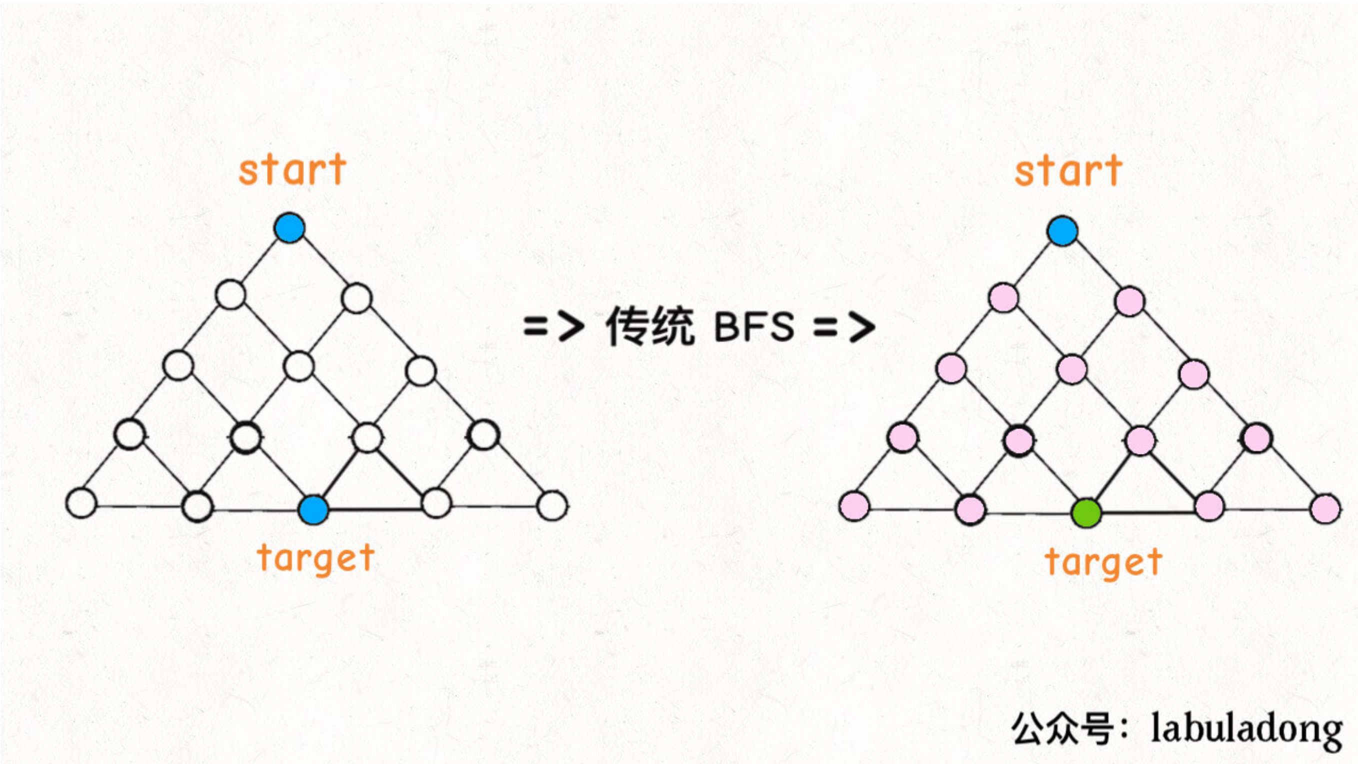
至此，我们就解决这道题目了。有一个比较小的优化：可以不需要 `dead` 这个哈希集合，可以直接将这些元素初始化到 `visited` 集合中，效果是一样的，可能更加优雅一些。

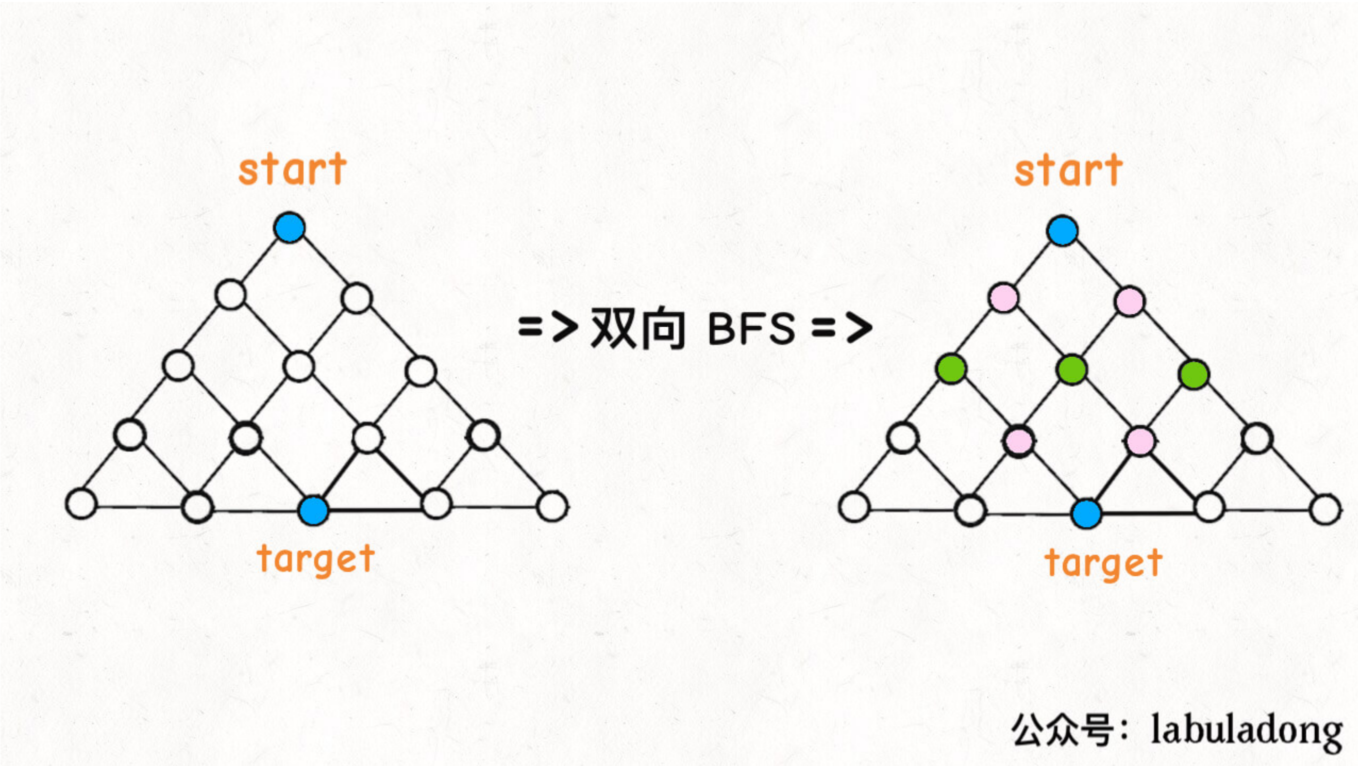
四、双向 BFS 优化

你以为到这里 BFS 算法就结束了？恰恰相反。BFS 算法还有一种稍微高级一点的优化思路：**双向 BFS**，可以进一步提高算法的效率。

篇幅所限，这里就提一下区别：**传统的 BFS 框架就是从起点开始向四周扩散，遇到终点时停止；而双向 BFS 则是从起点和终点同时开始扩散，当两边有交集的时候停止。**

为什么这样能够能够提升效率呢？其实从 Big O 表示法分析算法复杂度的话，它俩的最坏复杂度都是 `O(N)`，但是实际上双向 BFS 确实会快一些，我给你画两张图看一眼就明白了：





图示中的树形结构，如果终点在最底部，按照传统 BFS 算法的策略，会把整棵树的节点都搜索一遍，最后找到 target；而双向 BFS 其实只遍历了半棵树就出现了交集，也就是找到了最短距离。从这个例子可以直观地感受到，双向 BFS 是要比传统 BFS 高效的。

不过，双向 BFS 也有局限，因为你必须知道终点在哪里。比如我们刚才讨论的二叉树最小高度的问题，你一开始根本就不知道终点在哪里，也就无法使用双向 BFS；但是第二个密码锁的问题，是可以使用双向 BFS 算法来提高效率的，代码稍加修改即可：

```java
int openLock(String[] deadends, String target) {
    Set<String> deads = new HashSet<>();
    for (String s : deadends) deads.add(s);
    // 用集合不用队列，可以快速判断元素是否存在
    Set<String> q1 = new HashSet<>();
    Set<String> q2 = new HashSet<>();
    Set<String> visited = new HashSet<>();

    int step = 0;
    q1.add("0000");
    q2.add(target);

    while (!q1.isEmpty() && !q2.isEmpty()) {
        // 哈希集合在遍历的过程中不能修改，用 temp 存储扩散结果
        Set<String> temp = new HashSet<>();

        /* 将 q1 中的所有节点向周围扩散 */
        for (String cur : q1) {
            /* 判断是否到达终点 */
            if (deads.contains(cur))
                continue;
            if (q2.contains(cur))
                return step;
```

```
            visited.add(cur);

            /* 将一个节点的未遍历相邻节点加入集合 */
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                if (!visited.contains(up))
                    temp.add(up);
                String down = minusOne(cur, j);
                if (!visited.contains(down))
                    temp.add(down);
            }
        }
        /* 在这里增加步数 */
        step++;
        // temp 相当于 q1
        // 这里交换 q1 q2，下一轮 while 就是扩散 q2
        q1 = q2;
        q2 = temp;
    }
    return -1;
}
```

双向 BFS 还是遵循 BFS 算法框架的，只是**不再使用队列，而是使用 HashSet 方便快速判断两个集合是否有交集**。

另外的一个技巧点就是 **while 循环的最后交换 `q1` 和 `q2` 的内容**，所以只要默认扩散 `q1` 就相当于轮流扩散 `q1` 和 `q2`。

其实双向 BFS 还有一个优化，就是在 while 循环开始时做一个判断：

```
// ...
while (!q1.isEmpty() && !q2.isEmpty()) {
    if (q1.size() > q2.size()) {
        // 交换 q1 和 q2
        temp = q1;
        q1 = q2;
        q2 = temp;
    }
    // ...
```

为什么这是一个优化呢？

因为按照 BFS 的逻辑，队列（集合）中的元素越多，扩散之后新的队列（集合）中的元素就越多；在双向 BFS 算法中，如果我们每次都选择一个较小的集合进行扩散，那么占用的空间增长速度就会慢一些，效率就会高一些。

不过话说回来，**无论传统 BFS 还是双向 BFS，无论做不做优化，从 Big O 衡量标准来看，时间复杂度都是一样的**，只能说双向 BFS 是一种 trick，算法运行的速度会相对快一点，掌握不掌握其实都无所谓。最关键的是把 BFS 通用框架记下来，反正所有 BFS 算法都可以用它套出解法。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

——————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



如何用 BFS 算法秒杀各种智力题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

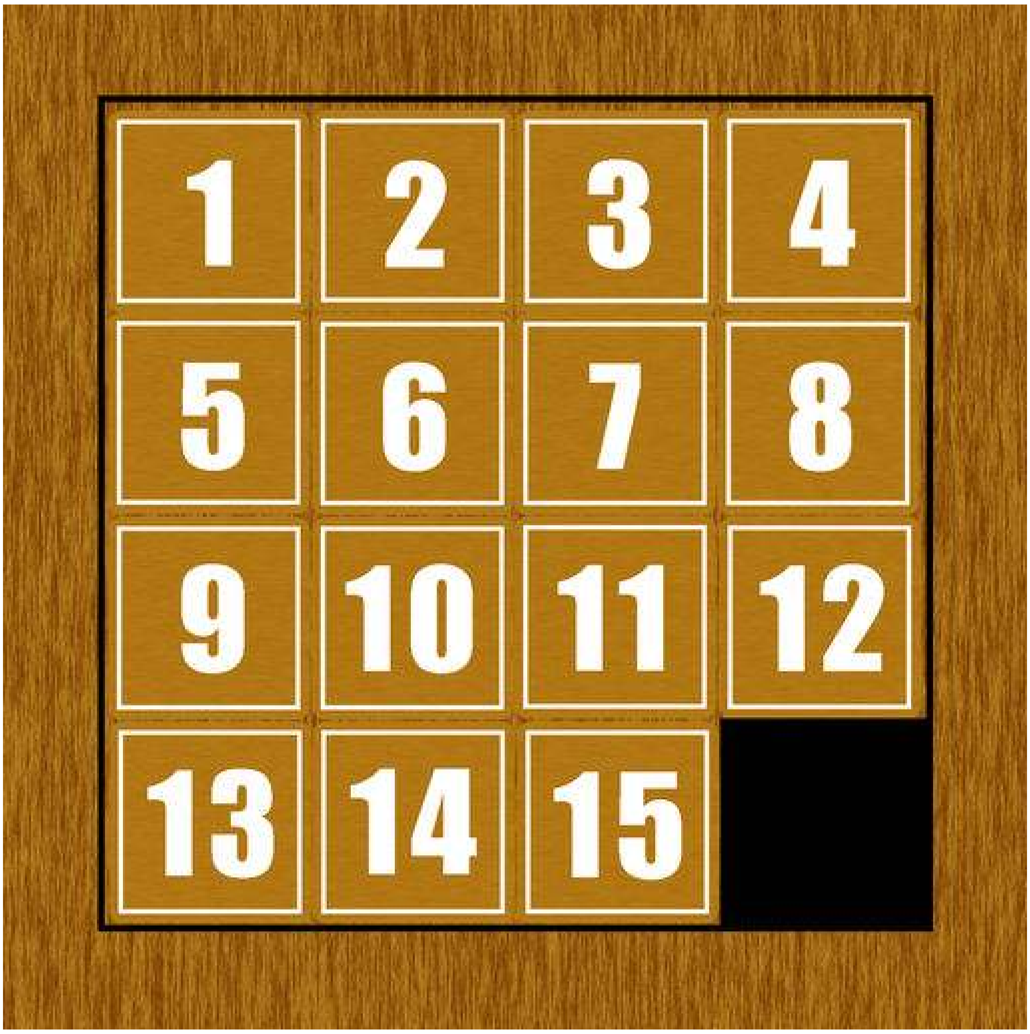
微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

773. 滑动谜题（困难）

滑动拼图游戏大家应该都玩过，下图是一个 4x4 的滑动拼图：



拼图中有一个格子是空的，可以利用这个空着的格子移动其他数字。你需要通过移动这些数字，得到某个特定排列顺序，这样就算赢了。

我小时候还玩过一款叫做「华容道」的益智游戏，也和滑动拼图比较类似：



实际上，滑动拼图游戏也叫数字华容道，你看它俩挺相似的。

那么这种游戏怎么玩呢？我记得是有一些套路的，类似于魔方还原公式。但是我们今天不来研究让人头秃的技巧，**这些益智游戏通通可以用暴力搜索算法解决，所以今天我们就学以致用，用 BFS 算法框架来秒杀这些游戏。**

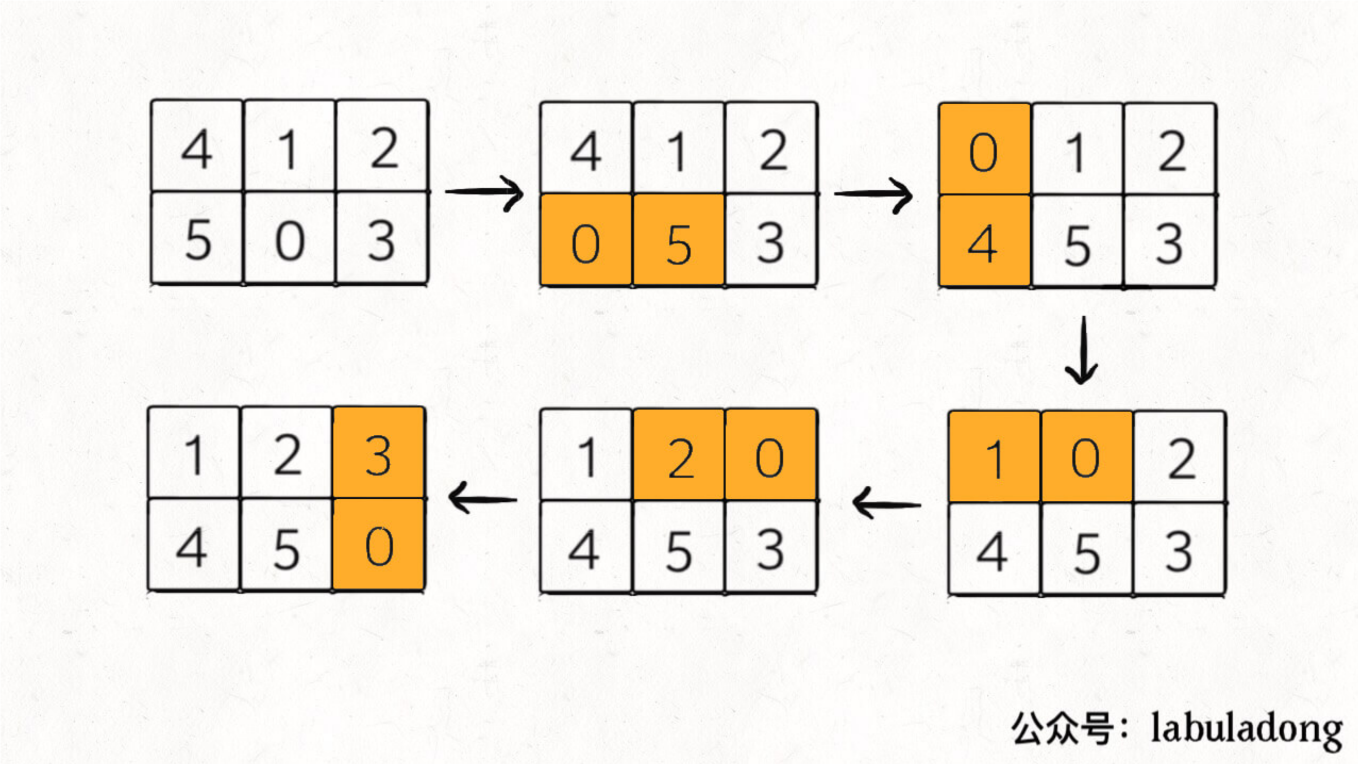
一、题目解析

力扣第 773 题「滑动谜题」就是这个问题，题目的要求如下：

给你一个 2x3 的滑动拼图，用一个 2x3 的数组 `board` 表示。拼图中有数字 0~5 六个数，其中**数字 0 就表示那个空着的格子**，你可以移动其中的数字，当 `board` 变为 `[[1,2,3],[4,5,0]]` 时，赢得游戏。

请你写一个算法，计算赢得游戏需要的最少移动次数，如果不能赢得游戏，返回 -1。

比如说输入的二维数组 `board = [[4,1,2],[5,0,3]]`，算法应该返回 5：



如果输入的是 `board = [[1,2,3],[5,4,0]]`，则算法返回 -1，因为这种局面下无论如何都不能赢得游戏。

二、思路分析

对于这种计算最小步数的问题，我们就要敏感地想到 BFS 算法。

这个题目转化成 BFS 问题是有一些技巧的，我们面临如下问题：

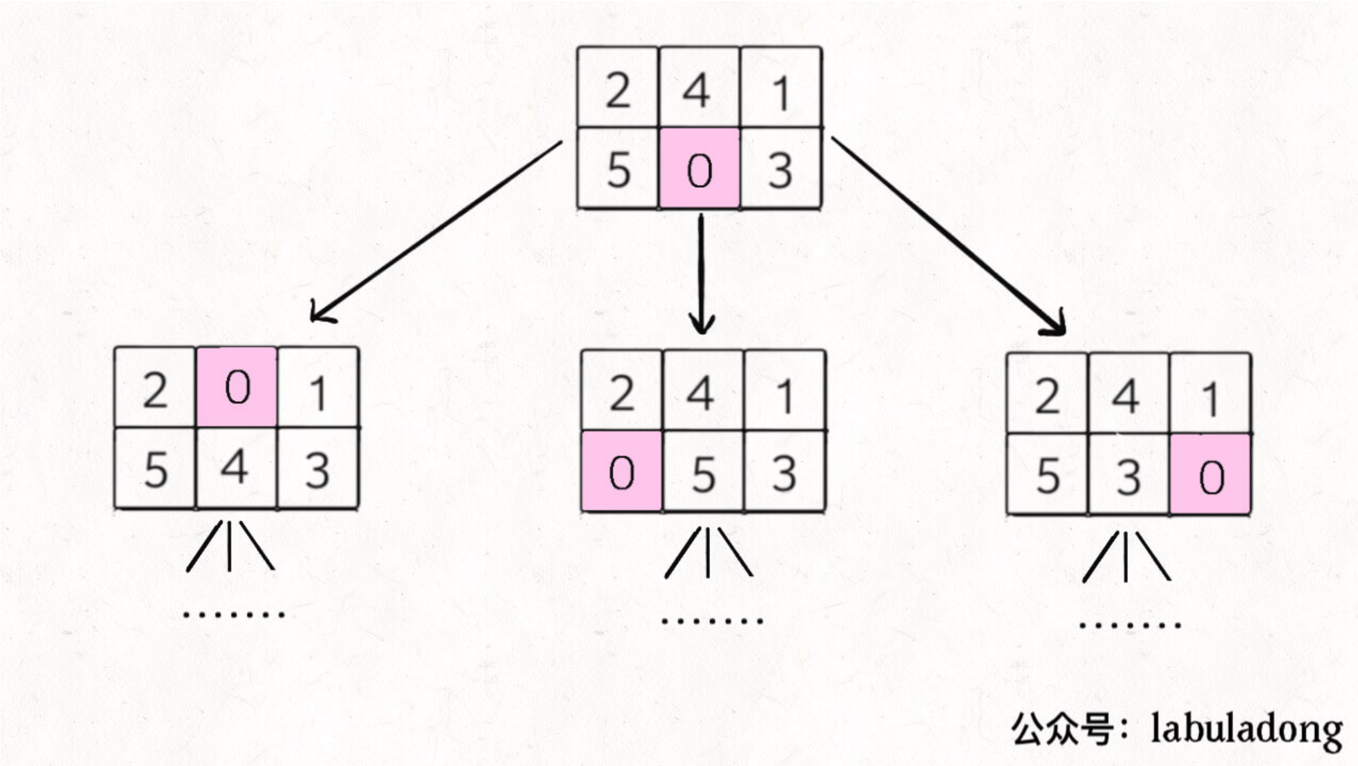
1、一般的 BFS 算法，是从一个起点 `start` 开始，向终点 `target` 进行寻路，但是拼图问题不是在寻路，而是在不断交换数字，这应该怎么转化成 BFS 算法问题呢？

2、即便这个问题能够转化成 BFS 问题，如何处理起点 `start` 和终点 `target`？它们都是数组哎，把数组放进队列，套 BFS 框架，想想就比较麻烦且低效。

首先回答第一个问题，**BFS 算法并不只是一个寻路算法，而是一种暴力搜索算法**，只要涉及暴力穷举的问题，BFS 就可以用，而且可以最快地找到答案。

你想想计算机怎么解决问题的？哪有那么多奇技淫巧，本质上就是把所有可行解暴力穷举出来，然后从中找到一个最优解罢了。

明白了这个道理，我们的问题就转化成了：**如何穷举出 `board` 当前局面下可能衍生出的所有局面**？这就简单了，看数字 0 的位置呗，和上下左右的数字进行交换就行了：



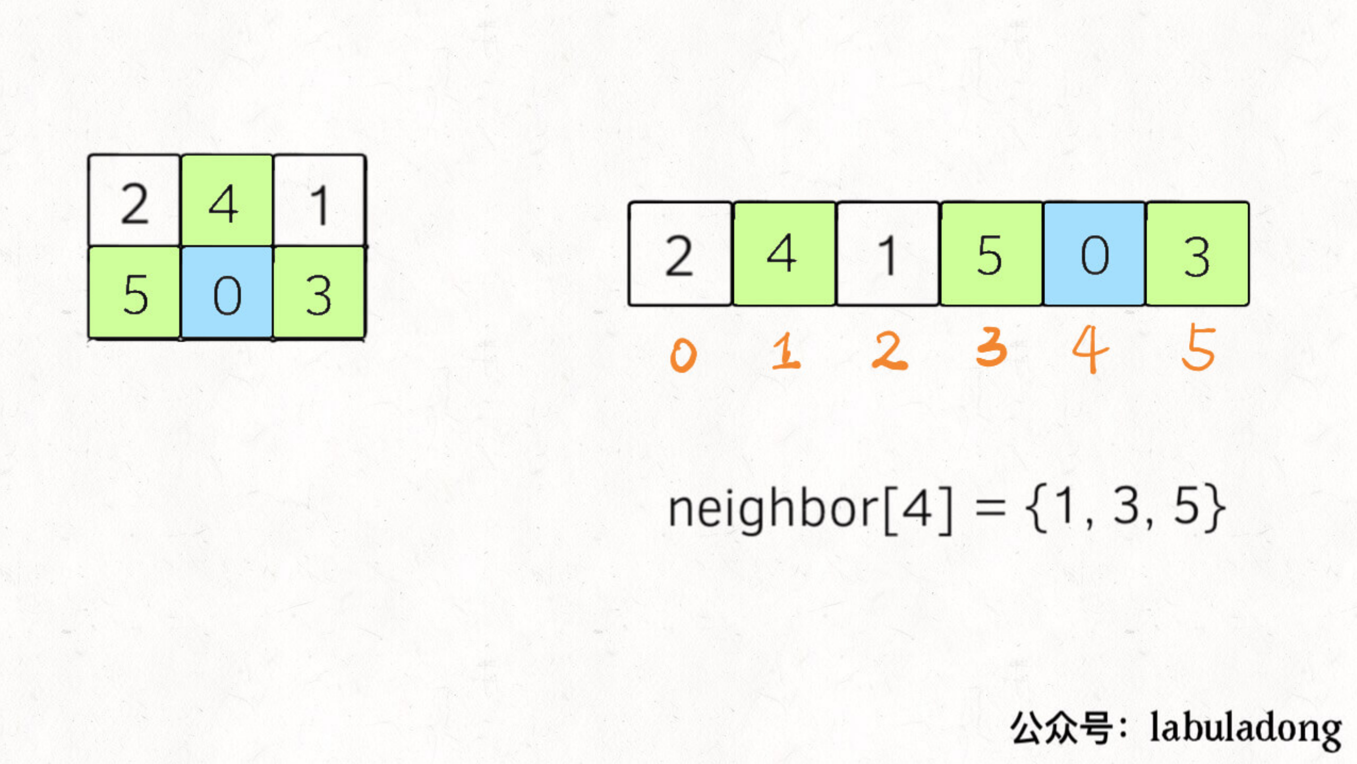
这样其实就是一个 BFS 问题，每次先找到数字 0，然后和周围的数字进行交换，形成新的局面加入队列……当第一次到达 `target` 时，就得到了赢得游戏的最少步数。

对于第二个问题，我们这里的 `board` 仅仅是 2x3 的二维数组，所以可以压缩成一个一维字符串。**其中比较有技巧性的点在于，二维数组有「上下左右」的概念，压缩成一维后，如何得到某一个索引上下左右的索引？**

很简单，我们只要手动写出来这个映射就行了：

```
// 记录一维字符串的相邻索引
int[][] neighbor = new int[][]{
        {1, 3},
        {0, 4, 2},
        {1, 5},
        {0, 4},
        {3, 1, 5},
        {4, 2}
};
```

这个含义就是，在一维字符串中，索引 `i` 在二维数组中的的相邻索引为 `neighbor[i]`：



至此，我们就把这个问题完全转化成标准的 BFS 问题了，借助前文 BFS 算法框架 的代码框架，直接就可以套出解法代码了：

```java
public int slidingPuzzle(int[][] board) {
    int m = 2, n = 3;
    StringBuilder sb = new StringBuilder();
    String target = "123450";
    // 将 2x3 的数组转化成字符串作为 BFS 的起点
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            sb.append(board[i][j]);
        }
    }
    String start = sb.toString();

    // 记录一维字符串的相邻索引
    int[][] neighbor = new int[][]{
            {1, 3},
            {0, 4, 2},
            {1, 5},
            {0, 4},
            {3, 1, 5},
            {4, 2}
    };

    /******* BFS 算法框架开始 *******/
    Queue<String> q = new LinkedList<>();
    HashSet<String> visited = new HashSet<>();
    // 从起点开始 BFS 搜索
    q.offer(start);
    visited.add(start);
```

```java
    int step = 0;
    while (!q.isEmpty()) {
        int sz = q.size();
        for (int i = 0; i < sz; i++) {
            String cur = q.poll();
            // 判断是否达到目标局面
            if (target.equals(cur)) {
                return step;
            }
            // 找到数字 0 的索引
            int idx = 0;
            for (; cur.charAt(idx) != '0'; idx++) ;
            // 将数字 0 和相邻的数字交换位置
            for (int adj : neighbor[idx]) {
                String new_board = swap(cur.toCharArray(), adj, idx);
                // 防止走回头路
                if (!visited.contains(new_board)) {
                    q.offer(new_board);
                    visited.add(new_board);
                }
            }
        }
        step++;
    }
    /******* BFS 算法框架结束 *******/
    return -1;
}

private String swap(char[] chars, int i, int j) {
    char temp = chars[i];
    chars[i] = chars[j];
    chars[j] = temp;
    return new String(chars);
}
```

至此，这道题目就解决了，其实框架完全没有变，套路都是一样的，我们只是花了比较多的时间将滑动拼图游戏转化成 BFS 算法。

很多益智游戏都是这样，虽然看起来特别巧妙，但都架不住暴力穷举，常用的算法就是回溯算法或者 BFS 算法。

拓展延伸

下面呢，我们讲一点进阶的数学知识，能够快速判断题目输入的滑动拼图是否有解，有兴趣的读者可以看一看。

我们在组合数学中学过「群论」，群论中讲到一类常见的群叫做「置换群」，置换群中有一个概念叫做「对换」。

之所以说这些专业术语，是方便有兴趣的读者去搜索学习，即便你没学过组合数学也没关系，你权当这是一种技巧，记下来就好了。

所谓「对换」，其实就是一个元组，比如 `(2, 5)`，其含义是：将元素 `2` 和元素 `5` 交换位置。

就比如题目给的 2 x 3 棋盘，如果输入为 `[[4,2,1],[5,0,3]]`，那么进行对换 `(2, 5)` 之后棋盘变成 `[[4,5,1],[2,0,3]]`。

进一步，「对换」可以相乘，多个对换的乘积代表连续进行元素交换。

比如输入为 `[[4,2,1],[5,0,3]]`，进行对换 `(2, 5)(4, 3)` 意味着将 `2` 和 `5` 交换，再将 `4` 和 `3` 交换位置，棋盘变成 `[[3,5,1],[2,0,4]]`。

接下来引入「奇置换」和「偶置换」的概念。

其实也很简单，所谓奇置换，就是奇数个「对换」相乘，所谓偶置换，就是偶数个「对换」相乘。

这里面就有一个重要特性了：奇置换和偶置换是不可能互相转化的，即一个奇置换产生的结果，不可能通过偶置换产生出来，反之亦然。

知道了这个规律，如何快速判断一个棋盘是否有解呢？

题目说，我们每次只能交换 `0` 和相邻的元素，最终需要把输入棋盘 `board` 变成 `target = [[1,2,3],[4,5,0]]`，那么我们可以分三步进行判断：

1、暂时忽略「只能交换 `0` 和相邻的元素」这个限制，你可以交换任意两个元素，这样可以得到一个「对换」序列 `A` 能够把 `board` 变成 `target`

2、在「只能交换 `0` 和相邻的元素」的限制下，可以得到一个「对换」序列 `B` 将 `0` 移动到目标位置（右下角）。

3、判断对换序列 `A` 和 `B` 的奇偶性，如果奇偶性相同，则必然有解，反之则必然无解。

举例来说，比如输入 `board = [[4,1,2],[5,0,3]]`，按照上述步骤，我们可以获得对换序列 `A` 和 `B`：

```
A = (4, 1)(4, 3)(4, 5)(5, 0)
B = (0, 3)
```

你看，`A` 是偶置换，`B` 是奇置换，说明没有办法将 `board` 转化成 `target`，不信你用代码跑一下，输出应该是 -1。

再举个有解的例子，如果输入 `board = [[5,0,3],[2,4,1]]`，按照上述步骤，我们可以获得对换序列 `A` 和 `B`：

```
A = (1, 5)(5, 0)(5, 2)(5, 4)
B = (0, 4)(0, 1)
```

你看，`A` 和 `B` 都是偶置换，所以肯定有办法将 `board` 转化成 `target`，你也可以用代码验证。

这个技巧就讲到这里，更深层次的原理是数学上的一些性质，所以这里就不多探讨了，有兴趣的读者可以根据关键词自行搜索学习。

如果你有高质量的问题，可以 [点击这里](#) 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



悟剑篇、动态规划



动态规划的底层逻辑也是穷举，只不过动态规划问题具有一些特殊的性质，使得穷举的过程中存在可优化的空间。

这里先提醒你，学习动态规划问题要格外注意这几个词：「**状态**」，「**选择**」，「**dp 数组的定义**」。你把这几个词理解到位了，就理解了动态规划的核心。

当然，动态规划问题的题型非常广泛，我不能保证你理解了核心就能做出所有动态规划题目，但我保证你理解了核心原理之后可以很轻松地理解别人的正确解法。如果自己勤加练习和总结，解决大部分中上难度的动态规划问题应该是没什么问题的。

公众号标签：[手把手刷动态规划](#)

4.1 动态规划核心原理

关于这一章的重要性，我觉得不需要再强调了。

字越少，重要性越高，相信你会时常回来温习本章的内容。

动态规划解题核心框架

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong



学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

509. 斐波那契数（简单）

322. 零钱兑换（中等）

本文有视频版：动态规划框架套路详解

这篇文章是我们公众号半年前一篇 200 多赞赏的成名之作 动态规划详解 的进阶版。由于账号迁移的原因，旧文无法被搜索到，所以我润色了本文，并添加了更多干货内容，希望本文成为解决动态规划的一部「指导方针」。

动态规划问题（Dynamic Programming）应该是很多读者头疼的，不过这类问题也是最具有技巧性，最有意思的。本书使用了整整一个章节专门来写这个算法，动态规划的重要性也可见一斑。

本文解决几个问题：

动态规划是什么？解决动态规划问题有什么技巧？如何学习动态规划？

刷题刷多了就会发现，算法技巧就那几个套路，**我们后续的动态规划系列章节，都在使用本文的解题框架思维**，如果你心里有数，就会轻松很多。所以本文放在第一章，来扒一扒动态规划的裤子，形成一套解决这类问题的思维框架，希望能够成为解决动态规划问题的一部指导方针。本文就来讲解该算法的基本套路框架，下面上干货。

首先，动态规划问题的一般形式就是求最值。动态规划其实是运筹学的一种最优化方法，只不过在计算机问题上应用比较多，比如说让你求**最长**递增子序列呀，**最小**编辑距离呀等等。

既然是要求最值，核心问题是什么呢？**求解动态规划的核心问题是穷举**。因为要求最值，肯定要把所有可行的答案穷举出来，然后在其中找最值呗。

动态规划这么简单，就是穷举就完事了？我看到的动态规划问题都很难啊！

首先，动态规划的穷举有点特别，因为这类问题**存在「重叠子问题」**，如果暴力穷举的话效率会极其低下，所以需要「备忘录」或者「DP table」来优化穷举过程，避免不必要的计算。

而且，动态规划问题一定会**具备「最优子结构」**，才能通过子问题的最值得到原问题的最值。

另外，虽然动态规划的核心思想就是穷举求最值，但是问题可以千变万化，穷举所有可行解其实并不是一件容易的事，只有列出**正确的「状态转移方程」**，才能正确地穷举。

以上提到的重叠子问题、最优子结构、状态转移方程就是动态规划三要素。具体什么意思等会会举例详解，但是在实际的算法问题中，**写出状态转移方程是最困难的**，这也就是为什么很多朋友觉得动态规划问题困难的原因，我来提供我研究出来的一个思维框架，辅助你思考状态转移方程：

明确 base case -> 明确「状态」-> 明确「选择」 -> 定义 dp 数组/函数的含义。

按上面的套路走，最后的结果就可以套这个框架：

```
# 初始化 base case
dp[0][0][...] = base
# 进行状态转移
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 求最值(选择1，选择2...)
```

下面通过斐波那契数列问题和凑零钱问题来详解动态规划的基本原理。前者主要是让你明白什么是重叠子问题（斐波那契数列没有求最值，所以严格来说不是动态规划问题），后者主要举集中于如何列出状态转移方程。

一、斐波那契数列

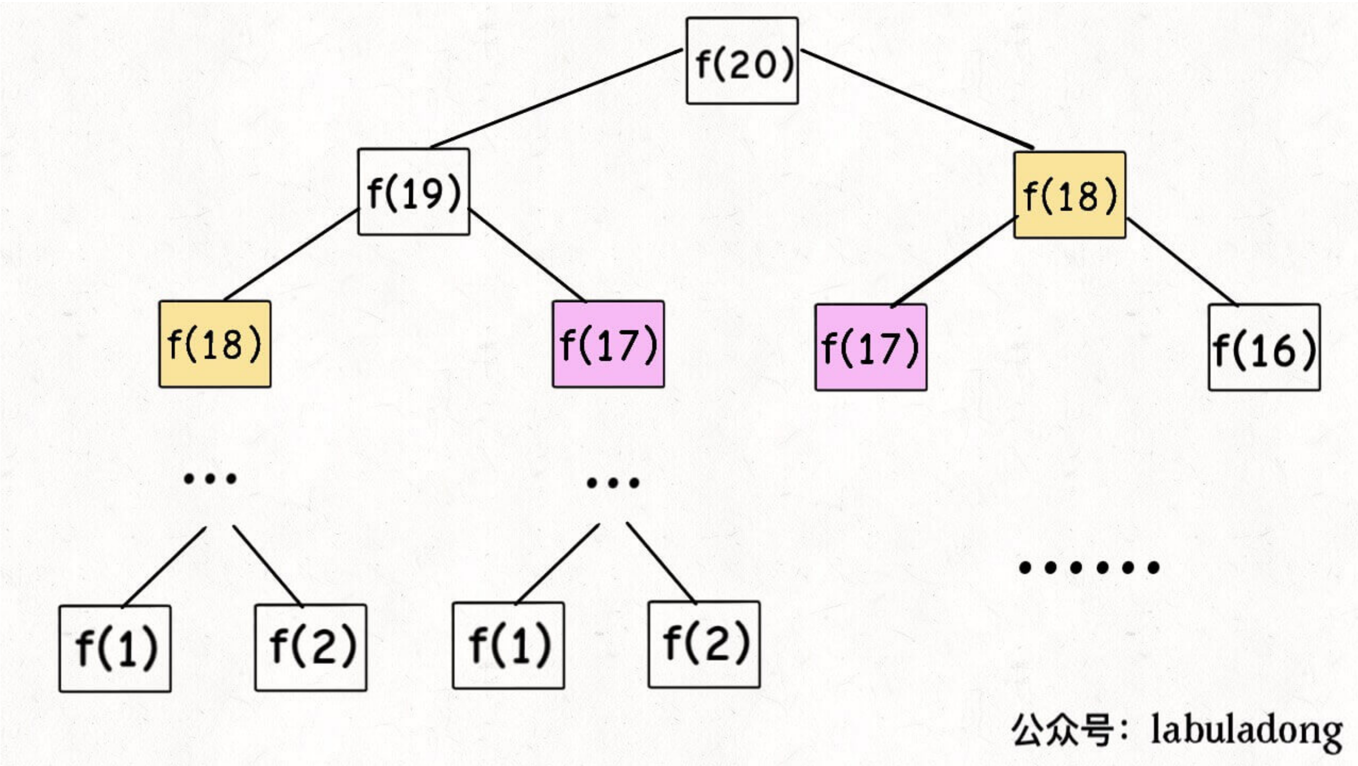
力扣第 509 题「斐波那契数」就是这个问题，请读者不要嫌弃这个例子简单，**只有简单的例子才能让你把精力充分集中在算法背后的通用思想和技巧上，而不会被那些隐晦的细节问题搞的莫名其妙**。想要困难的例子，历史文章里有的是。

1、暴力递归

斐波那契数列的数学形式就是递归的，写成代码就是这样：

```
int fib(int N) {
    if (N == 1 || N == 2) return 1;
    return fib(N - 1) + fib(N - 2);
}
```

这个不用多说了，学校老师讲递归的时候似乎都是拿这个举例。我们也知道这样写代码虽然简洁易懂，但是十分低效，低效在哪里？假设 n = 20，请画出递归树：



> PS：但凡遇到需要递归的问题，最好都画出递归树，这对你分析算法的复杂度，寻找算法低效的原因都有巨大帮助。

这个递归树怎么理解？就是说想要计算原问题 `f(20)`，我就得先计算出子问题 `f(19)` 和 `f(18)`，然后要计算 `f(19)`，我就要先算出子问题 `f(18)` 和 `f(17)`，以此类推。最后遇到 `f(1)` 或者 `f(2)` 的时候，结果已知，就能直接返回结果，递归树不再向下生长了。

递归算法的时间复杂度怎么计算？就是用子问题个数乘以解决一个子问题需要的时间。

首先计算子问题个数，即递归树中节点的总数。显然二叉树节点总数为指数级别，所以子问题个数为 O(2^n)。

然后计算解决一个子问题的时间，在本算法中，没有循环，只有 `f(n - 1) + f(n - 2)` 一个加法操作，时间为 O(1)。

所以，这个算法的时间复杂度为二者相乘，即 O(2^n)，指数级别，爆炸。

观察递归树，很明显发现了算法低效的原因：存在大量重复计算，比如 `f(18)` 被计算了两次，而且你可以看到，以 `f(18)` 为根的这个递归树体量巨大，多算一遍，会耗费巨大的时间。更何况，还不止 `f(18)` 这一个节点被重复计算，所以这个算法及其低效。

这就是动态规划问题的第一个性质：**重叠子问题**。下面，我们想办法解决这个问题。

2、带备忘录的递归解法

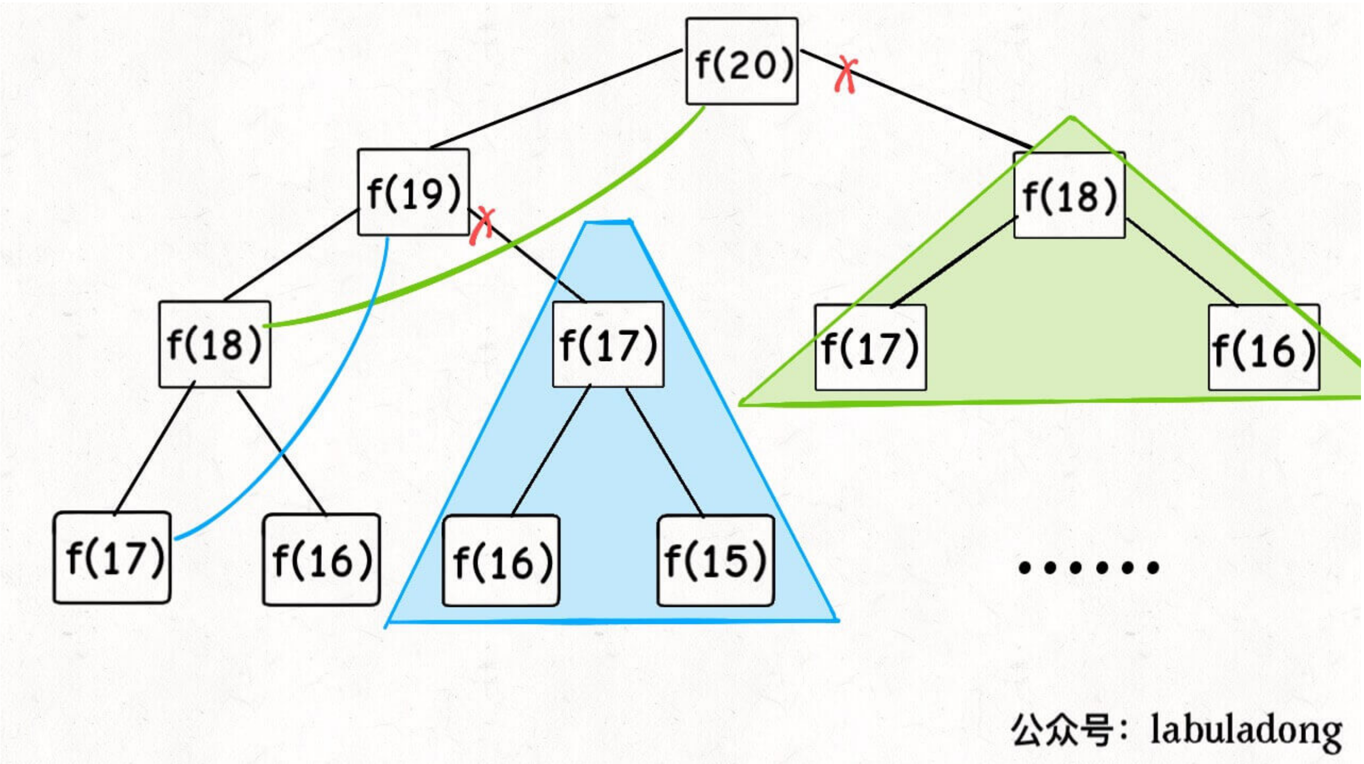
明确了问题，其实就已经把问题解决了一半。即然耗时的原因是重复计算，那么我们可以造一个「备忘录」，每次算出某个子问题的答案后别急着返回，先记到「备忘录」里再返回；每次遇到一个子问题先去「备忘录」里查一查，如果发现之前已经解决过这个问题了，直接把答案拿出来用，不要再耗时去计算了。

一般使用一个数组充当这个「备忘录」，当然你也可以使用哈希表（字典），思想都是一样的。

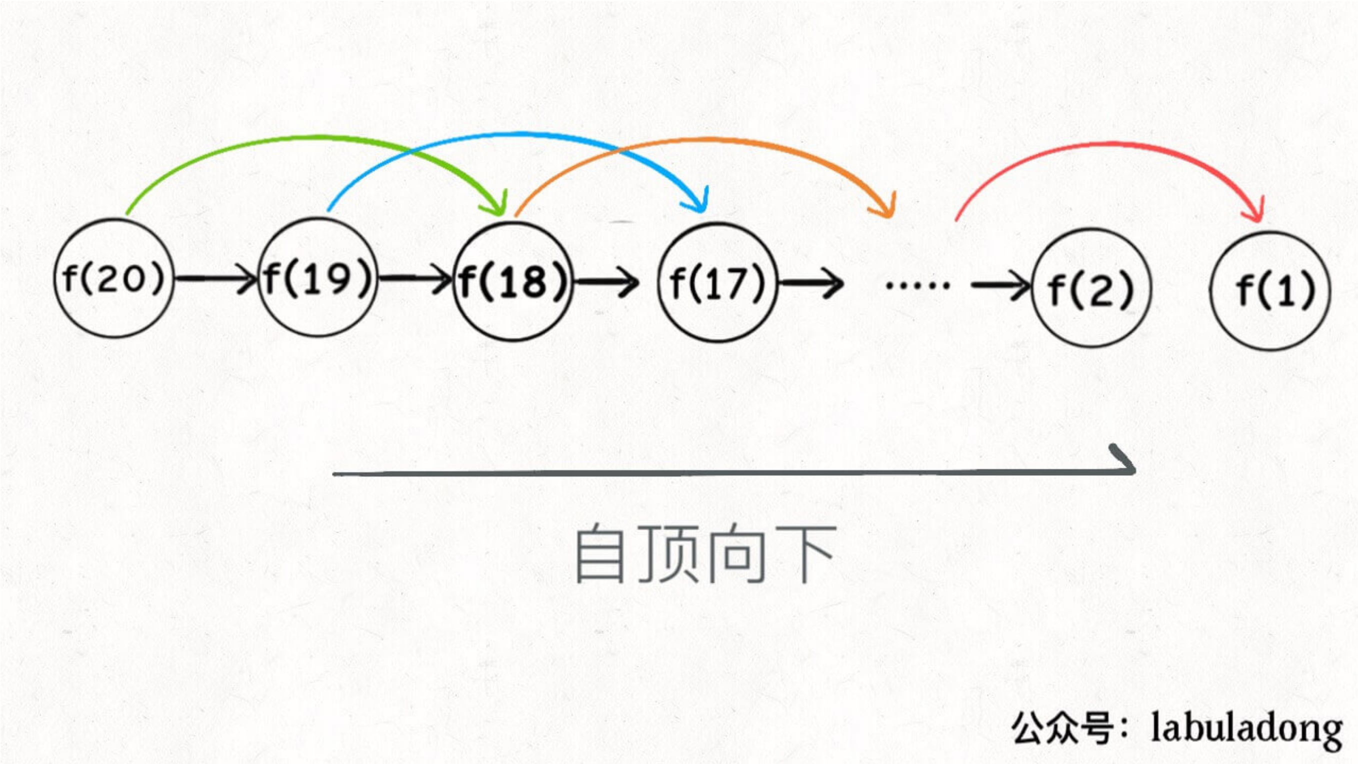
```java
int fib(int N) {
    // 备忘录全初始化为 0
    int[] memo = new int[N + 1];
    // 进行带备忘录的递归
    return helper(memo, N);
}

int helper(int[] memo, int n) {
    // base case
    if (n == 0 || n == 1) return n;
    // 已经计算过，不用再计算了
    if (memo[n] != 0) return memo[n];
    memo[n] = helper(memo, n - 1) + helper(memo, n - 2);
    return memo[n];
}
```

现在，画出递归树，你就知道「备忘录」到底做了什么。



实际上，带「备忘录」的递归算法，把一棵存在巨量冗余的递归树通过「剪枝」，改造成了一幅不存在冗余的递归图，极大减少了子问题（即递归图中节点）的个数。



递归算法的时间复杂度怎么计算？就是用子问题个数乘以解决一个子问题需要的时间。

子问题个数，即图中节点的总数，由于本算法不存在冗余计算，子问题就是 `f(1)`, `f(2)`, `f(3)` ... `f(20)`，数量和输入规模 n = 20 成正比，所以子问题个数为 O(n)。

解决一个子问题的时间，同上，没有什么循环，时间为 O(1)。

所以，本算法的时间复杂度是 O(n)，比起暴力算法，是降维打击。

至此，带备忘录的递归解法的效率已经和迭代的动态规划解法一样了。实际上，这种解法和常见的动态规划解法已经差不多了，只不过这种解法是「自顶向下」进行「递归」求解，我们更常见的动态规划代码是「自底向上」进行「递推」求解。

啥叫「自顶向下」？注意我们刚才画的递归树（或者说图），是从上向下延伸，都是从一个规模较大的原问题比如说 `f(20)`，向下逐渐分解规模，直到 `f(1)` 和 `f(2)` 这两个 base case，然后逐层返回答案，这就叫「自顶向下」。

啥叫「自底向上」？反过来，我们直接从最底下、最简单、问题规模最小、已知结果的 `f(1)` 和 `f(2)`（base case）开始往上推，直到推到我们想要的答案 `f(20)`。这就是「递推」的思路，这也是动态规划一般都脱离了递归，而是由循环迭代完成计算的原因。

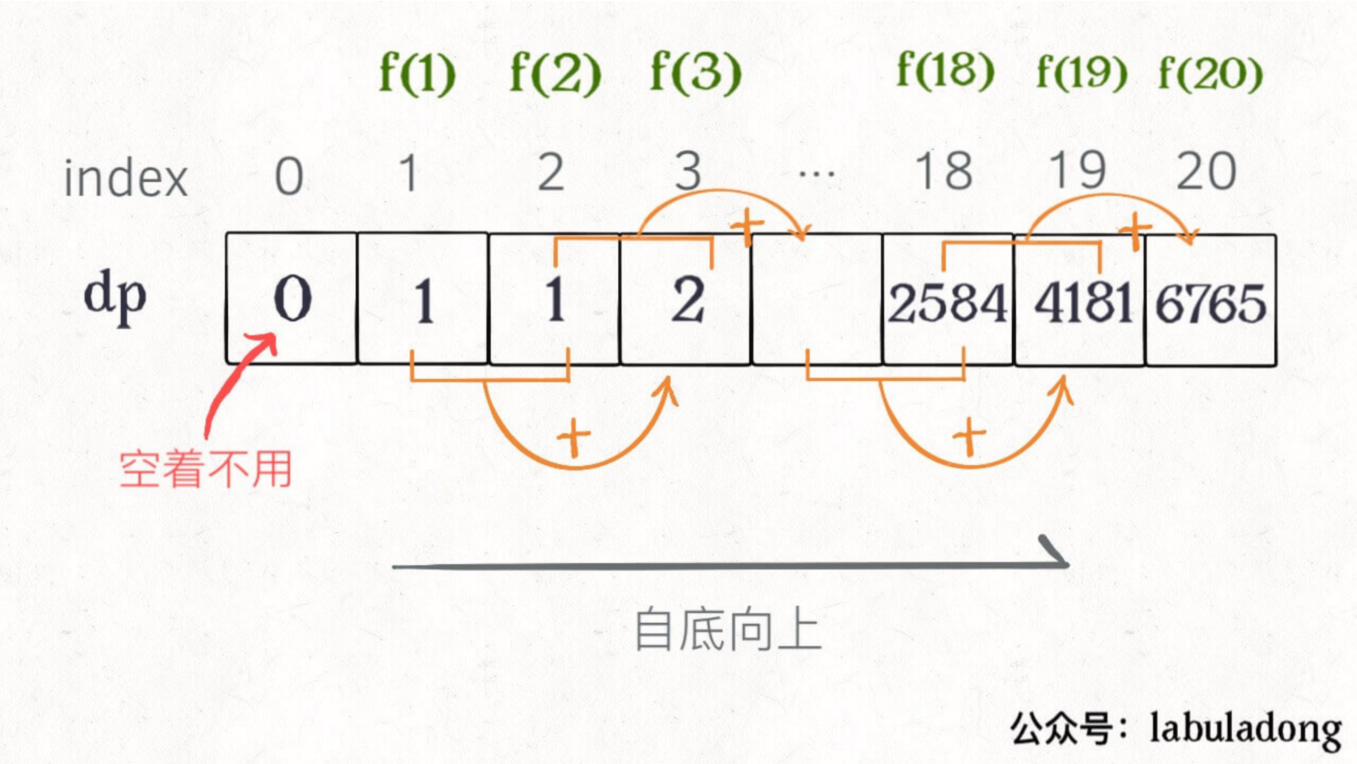
3、`dp` 数组的迭代（递推）解法

有了上一步「备忘录」的启发，我们可以把这个「备忘录」独立出来成为一张表，通常叫做 DP table，在这张表上完成「自底向上」的推算岂不美哉！

```
int fib(int N) {
    if (N == 0) return 0;
    int[] dp = new int[N + 1];
    // base case
    dp[0] = 0; dp[1] = 1;
```

```
    // 状态转移
    for (int i = 2; i <= N; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    return dp[N];
}
```



画个图就很好理解了，而且你发现这个 DP table 特别像之前那个「剪枝」后的结果，只是反过来算而已。实际上，带备忘录的递归解法中的「备忘录」，最终完成后就是这个 DP table，所以说这两种解法其实是差不多的，大部分情况下，效率也基本相同。

这里，引出「状态转移方程」这个名词，实际上就是描述问题结构的数学形式：

$$f(n) = \begin{cases} 1, n = 1, 2 \\ f(n-1) + f(n-2), n > 2 \end{cases}$$

为啥叫「状态转移方程」？其实就是为了听起来高端。

`f(n)` 的函数参数会不断变化，所以你把参数 `n` 想做一个状态，这个状态 `n` 是由状态 `n - 1` 和状态 `n - 2` 转移（相加）而来，这就叫状态转移，仅此而已。

你会发现，上面的几种解法中的所有操作，例如 `return f(n - 1) + f(n - 2)`，`dp[i] = dp[i - 1] + dp[i - 2]`，以及对备忘录或 DP table 的初始化操作，都是围绕这个方程式的不同表现形式。

可见列出「状态转移方程」的重要性，它是解决问题的核心，而且很容易发现，其实状态转移方程直接代表着暴力解法。

千万不要看不起暴力解，动态规划问题最困难的就是写出这个暴力解，即状态转移方程。

只要写出暴力解，优化方法无非是用备忘录或者 DP table，再无奥妙可言。

这个例子的最后，讲一个细节优化。

细心的读者会发现，根据斐波那契数列的状态转移方程，当前状态只和之前的两个状态有关，其实并不需要那么长的一个 DP table 来存储所有的状态，只要想办法存储之前的两个状态就行了。

所以，可以进一步优化，把空间复杂度降为 O(1)。这也就是我们最常见的计算斐波那契数的算法：

```
int fib(int n) {
    if (n == 0 || n == 1) {
        // base case
        return n;
    }
    // 分别代表 dp[i - 1] 和 dp[i - 2]
    int dp_i_1 = 1, dp_i_2 = 0;
    for (int i = 2; i <= n; i++) {
        // dp[i] = dp[i - 1] + dp[i - 2];
        int dp_i = dp_i_1 + dp_i_2;
        // 滚动更新
        dp_i_2 = dp_i_1;
        dp_i_1 = dp_i;
    }
    return dp_i_1;
}
```

这一般是动态规划问题的最后一步优化，如果我们发现每次状态转移只需要 DP table 中的一部分，那么可以尝试缩小 DP table 的大小，只记录必要的数据，从而降低空间复杂度。

上述例子就相当于把DP table 的大小从 `n` 缩小到 2。后续的动态规划章节中我们还会看到这样的例子，一般来说是把一个二维的 DP table 压缩成一维，即把空间复杂度从 O(n^2) 压缩到 O(n)。

有人会问，动态规划的另一个重要特性「最优子结构」，怎么没有涉及？下面会涉及。斐波那契数列的例子严格来说不算动态规划，因为没有涉及求最值，以上旨在说明重叠子问题的消除方法，演示得到最优解法逐步求精的过程。下面，看第二个例子，凑零钱问题。

二、凑零钱问题

这是力扣第 322 题「零钱兑换」：

给你 `k` 种面值的硬币，面值分别为 `c1, c2 ... ck`，每种硬币的数量无限，再给一个总金额 `amount`，问你**最少**需要几枚硬币凑出这个金额，如果不可能凑出，算法返回 -1 。算法的函数签名如下：

```
// coins 中是可选硬币面值，amount 是目标金额
int coinChange(int[] coins, int amount);
```

比如说 `k = 3`，面值分别为 1，2，5，总金额 `amount = 11`。那么最少需要 3 枚硬币凑出，即 11 = 5 + 5 + 1。

你认为计算机应该如何解决这个问题？显然，就是把所有可能的凑硬币方法都穷举出来，然后找找看最少需要多少枚硬币。

1、暴力递归

首先，这个问题是动态规划问题，因为它具有「最优子结构」的。**要符合「最优子结构」，子问题间必须互相独立**。啥叫相互独立？你肯定不想看数学证明，我用一个直观的例子来讲解。

比如说，假设你考试，每门科目的成绩都是互相独立的。你的原问题是考出最高的总成绩，那么你的子问题就是要把语文考到最高，数学考到最高…… 为了每门课考到最高，你要把每门课相应的选择题分数拿到最高，填空题分数拿到最高…… 当然，最终就是你每门课都是满分，这就是最高的总成绩。

得到了正确的结果：最高的总成绩就是总分。因为这个过程符合最优子结构，"每门科目考到最高"这些子问题是互相独立，互不干扰的。

但是，如果加一个条件：你的语文成绩和数学成绩会互相制约，数学分数高，语文分数就会降低，反之亦然。这样的话，显然你能考到的最高总成绩就达不到总分了，按刚才那个思路就会得到错误的结果。因为子问题并不独立，语文数学成绩无法同时最优，所以最优子结构被破坏。

回到凑零钱问题，为什么说它符合最优子结构呢？比如你想求 `amount = 11` 时的最少硬币数（原问题），如果你知道凑出 `amount = 10` 的最少硬币数（子问题），你只需要把子问题的答案加一（再选一枚面值为 1 的硬币）就是原问题的答案。因为硬币的数量是没有限制的，所以子问题之间没有相互制，是互相独立的。

> PS：关于最优子结构的问题，后文动态规划答疑篇 还会再举例探讨。

那么，既然知道了这是个动态规划问题，就要思考**如何列出正确的状态转移方程**？

1、**确定 base case**，这个很简单，显然目标金额 `amount` 为 0 时算法返回 0，因为不需要任何硬币就已经凑出目标金额了。

2、**确定「状态」，也就是原问题和子问题中会变化的变量**。由于硬币数量无限，硬币的面额也是题目给定的，只有目标金额会不断地向 base case 靠近，所以唯一的「状态」就是目标金额 `amount`。

3、**确定「选择」，也就是导致「状态」产生变化的行为**。目标金额为什么变化呢，因为你在选择硬币，你每选择一枚硬币，就相当于减少了目标金额。所以说所有硬币的面值，就是你的「选择」。

4、**明确 `dp` 函数/数组的定义**。我们这里讲的是自顶向下的解法，所以会有一个递归的 `dp` 函数，一般来说函数的参数就是状态转移中会变化的量，也就是上面说到的「状态」；函数的返回值就是题目要求我们计算的量。就本题来说，状态只有一个，即「目标金额」，题目要求我们计算凑出目标金额所需的最少硬币数量。

所以我们可以这样定义 `dp` 函数：`dp(n)` 表示，输入一个目标金额 `n`，返回凑出目标金额 `n` 所需的最少硬币数量。

搞清楚上面这几个关键点，解法的伪码就可以写出来了：

```
// 伪码框架
int coinChange(int[] coins, int amount) {
```

```
    // 题目要求的最终结果是 dp(amount)
    return dp(coins, amount)
}

// 定义：要凑出金额 n，至少要 dp(coins, n) 个硬币
int dp(int[] coins, int n) {
    // 做选择，选择需要硬币最少的那个结果
    for (int coin : coins) {
        res = min(res, 1 + dp(n - coin))
    }
    return res
}
```

根据伪码，我们加上 base case 即可得到最终的答案。显然目标金额为 0 时，所需硬币数量为 0；当目标金额小于 0 时，无解，返回 -1：

```
int coinChange(int[] coins, int amount) {
    // 题目要求的最终结果是 dp(amount)
    return dp(coins, amount)
}

int dp(int[] coins, int amount) {
    // base case
    if (amount == 0) return 0;
    if (amount < 0) return -1;

    int res = Integer.MAX_VALUE;
    for (int coin : coins) {
        // 计算子问题的结果
        int subProblem = dp(coins, amount - coin);
        // 子问题无解则跳过
        if (subProblem == -1) continue;
        // 在子问题中选择最优解，然后加一
        res = Math.min(res, subProblem + 1);
    }

    return res == Integer.MAX_VALUE ? -1 : res;
}
```

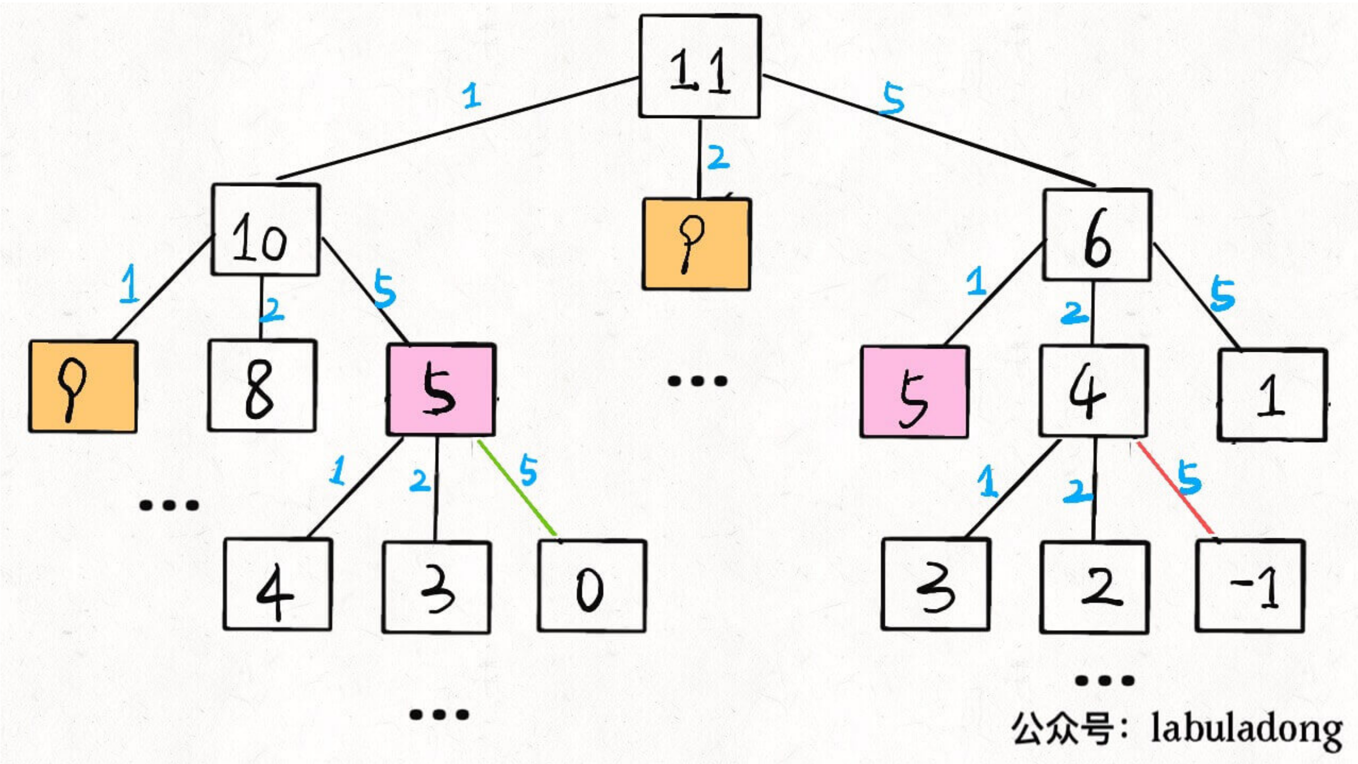
> PS：这里 `coinChange` 和 `dp` 函数的签名完全一样，所以理论上不需要额外写一个 `dp` 函数。但为了后文讲解方便，这里还是另写一个 `dp` 函数来实现主要逻辑。

> 另外，我经常看到有人问，子问题的结果为什么要加 1（`subProblem + 1`），而不是加硬币金额之类的。我这里统一提示一下，动态规划问题的关键是 `dp` 函数/数组的定义，你这个函数的返回值代表什么？你回过头去搞清楚这一点，然后就知道为什么要给子问题的返回值加一了。

至此，状态转移方程其实已经完成了，以上算法已经是暴力解法了，以上代码的数学形式就是状态转移方程：

$$dp(n) = \begin{cases} 0, n = 0 \\ -1, n < 0 \\ min\{dp(n - coin) + 1 | coin \in coins\}, n > 0 \end{cases}$$

至此，这个问题其实就解决了，只不过需要消除一下重叠子问题，比如 `amount = 11, coins = {1,2,5}` 时画出递归树看看：



递归算法的时间复杂度分析：子问题总数 x 解决每个子问题所需的时间。

子问题总数为递归树节点个数，这个比较难看出来，是 O(n^k)，总之只要是树形结构，节点个数必然是是指数级别的。每个子问题中含有一个 for 循环，复杂度为 O(k)。所以总时间复杂度为 O(k * n^k)，指数级别。

2、带备忘录的递归

类似之前斐波那契数列的例子，只需要稍加修改，就可以通过备忘录消除子问题：

```
int[] memo;

int coinChange(int[] coins, int amount) {
    memo = new int[amount + 1];
    // dp 数组全都初始化为特殊值
    Arrays.fill(memo, -666);

    return dp(coins, amount);
}

int dp(int[] coins, int amount) {
    if (amount == 0) return 0;
```

```
    if (amount < 0) return -1;
    // 查备忘录，防止重复计算
    if (memo[amount] != -666)
        return memo[amount];

    int res = Integer.MAX_VALUE;
    for (int coin : coins) {
        // 计算子问题的结果
        int subProblem = dp(coins, amount - coin);
        // 子问题无解则跳过
        if (subProblem == -1) continue;
        // 在子问题中选择最优解，然后加一
        res = Math.min(res, subProblem + 1);
    }
    // 把计算结果存入备忘录
    memo[amount] = (res == Integer.MAX_VALUE) ? -1 : res;
    return memo[amount];
}
```

不画图了，很显然「备忘录」大大减小了子问题数目，完全消除了子问题的冗余，所以子问题总数不会超过金额数 n，即子问题数目为 O(n)。处理一个子问题的时间不变，仍是 O(k)，所以总的时间复杂度是 O(kn)。

3、dp 数组的迭代解法

当然，我们也可以自底向上使用 dp table 来消除重叠子问题，关于「状态」「选择」和 base case 与之前没有区别，dp 数组的定义和刚才 dp 函数类似，也是把「状态」，也就是目标金额作为变量。不过 dp 函数体现在函数参数，而 dp 数组体现在数组索引：

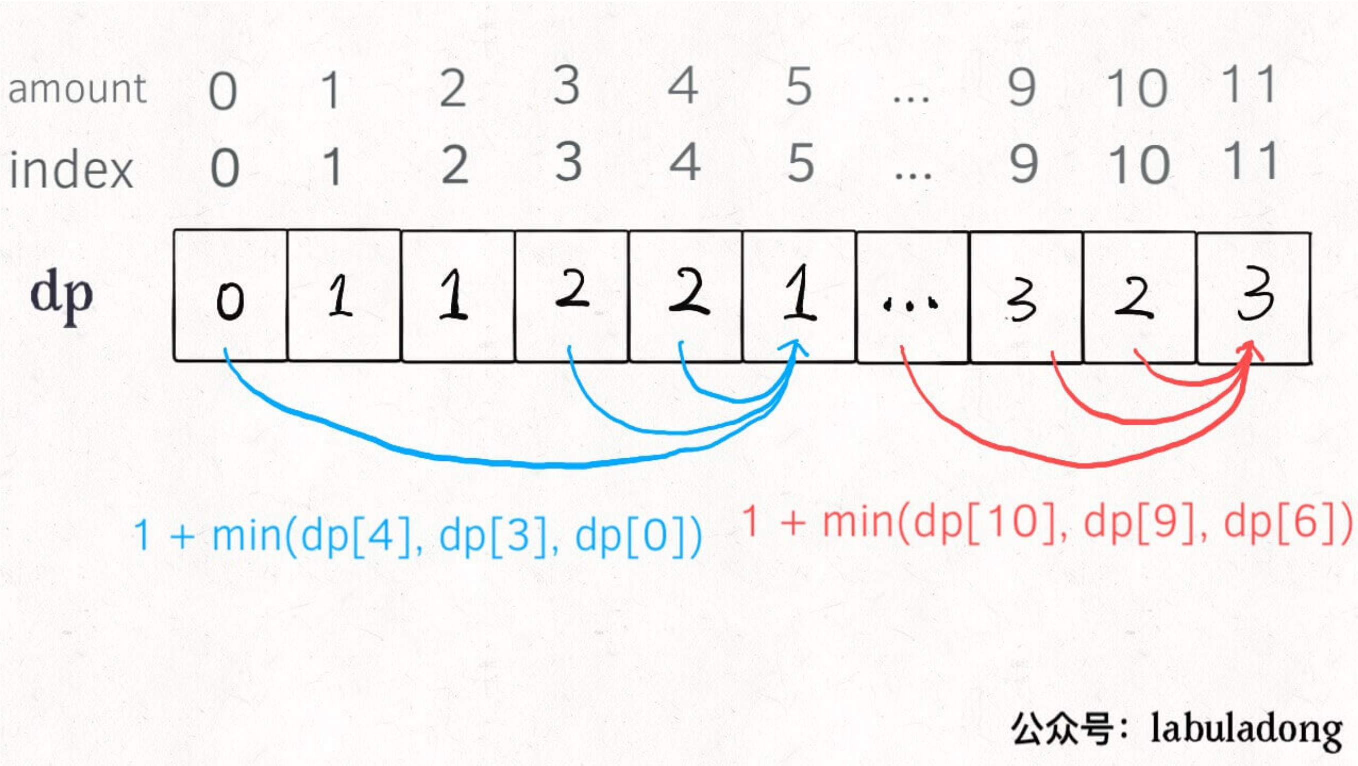
dp 数组的定义：当目标金额为 i 时，至少需要 dp[i] 枚硬币凑出。

根据我们文章开头给出的动态规划代码框架可以写出如下解法：

```
int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    // 数组大小为 amount + 1，初始值也为 amount + 1
    Arrays.fill(dp, amount + 1);

    // base case
    dp[0] = 0;
    // 外层 for 循环在遍历所有状态的所有取值
    for (int i = 0; i < dp.length; i++) {
        // 内层 for 循环在求所有选择的最小值
        for (int coin : coins) {
            // 子问题无解，跳过
            if (i - coin < 0) {
                continue;
            }
            dp[i] = Math.min(dp[i], 1 + dp[i - coin]);
        }
    }
```

```
    return (dp[amount] == amount + 1) ? -1 : dp[amount];
}
```



> PS：为啥 `dp` 数组初始化为 `amount + 1` 呢，因为凑成 `amount` 金额的硬币数最多只可能等于 `amount`（全用 1 元面值的硬币），所以初始化为 `amount + 1` 就相当于初始化为正无穷，便于后续取最小值。为啥不直接初始化为 int 型的最大值 `Integer.MAX_VALUE` 呢？因为后面有 `dp[i - coin] + 1`，这就会导致整型溢出。

三、最后总结

第一个斐波那契数列的问题，解释了如何通过「备忘录」或者「dp table」的方法来优化递归树，并且明确了这两种方法本质上是一样的，只是自顶向下和自底向上的不同而已。

第二个凑零钱的问题，展示了如何流程化确定「状态转移方程」，只要通过状态转移方程写出暴力递归解，剩下的也就是优化递归树，消除重叠子问题而已。

如果你不太了解动态规划，还能看到这里，真得给你鼓掌，相信你已经掌握了这个算法的设计技巧。

计算机解决问题其实没有任何奇技淫巧，它唯一的解决办法就是穷举，穷举所有可能性。算法设计无非就是先思考"如何穷举"，然后再追求"如何聪明地穷举"。

列出状态转移方程，就是在解决"如何穷举"的问题。之所以说它难，一是因为很多穷举需要递归实现，二是因为有的问题本身的解空间复杂，不那么容易穷举完整。

备忘录、DP table 就是在追求"如何聪明地穷举"。用空间换时间的思路，是降低时间复杂度的不二法门，除此之外，试问，还能玩出啥花活？

之后我们会有一章专门讲解动态规划问题，如果有任何问题都可以随时回来重读本文，希望读者在阅读每个题目和解法时，多往「状态」和「选择」上靠，才能对这套框架产生自己的理解，运用自如。

接下来可阅读：

- 动态规划设计：最长递增子序列

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



base case 和备忘录的初始值怎么定？

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

931. 下降路径最小和（中等）

之前我抽空看了以前文章的留言，很多读者对动态规划问题的 base case、备忘录初始值等问题存在疑问。

本文就专门讲一讲这类问题，顺便聊一聊怎么通过题目的蛛丝马迹揣测出题人的小心思，辅助我们解题。

看下力扣第 931 题「下降路径最小和」，输入为一个 `n * n` 的二维数组 `matrix`，请你计算从第一行落到最后一行，经过的路径和最小为多少。

函数签名如下：

```
int minFallingPathSum(int[][] matrix);
```

就是说你可以站在 `matrix` 的第一行的任意一个元素，需要下降到最后一行。

每次下降，可以向下、向左下、向右下三个方向移动一格。也就是说，可以从 `matrix[i][j]` 降到 `matrix[i+1][j]` 或 `matrix[i+1][j-1]` 或 `matrix[i+1][j+1]` 三个位置。

请你计算下降的「最小路径和」，比如说题目给了一个例子：

示例 1：

```
输入：matrix = [[2,1,3],[6,5,4],[7,8,9]]
输出：13
解释：下面是两条和最小的下降路径，用加粗标注：
[[2,1,3],      [[2,1,3],
  [6,5,4],       [6,5,4],
  [7,8,9]]       [7,8,9]]
```

我们前文写过两道「路径和」相关的文章：动态规划之最小路径和 和 用动态规划算法通关魔塔。

今天这道题也是类似的，不算是困难的题目，所以**我们借这道题来讲讲 base case 的返回值、备忘录的初始值、索引越界情况的返回值如何确定。**

不过还是要通过 动态规划的标准套路 介绍一下这道题的解题思路，首先我们可以定义一个 `dp` 数组：

```
int dp(int[][] matrix, int i, int j);
```

这个 `dp` 函数的含义如下：

从第一行（`matrix[0][..]`）向下落，落到位置 `matrix[i][j]` 的最小路径和为 `dp(matrix, i, j)`。

根据这个定义，我们可以把主函数的逻辑写出来：

```
int minFallingPathSum(int[][] matrix) {
    int n = matrix.length;
    int res = Integer.MAX_VALUE;

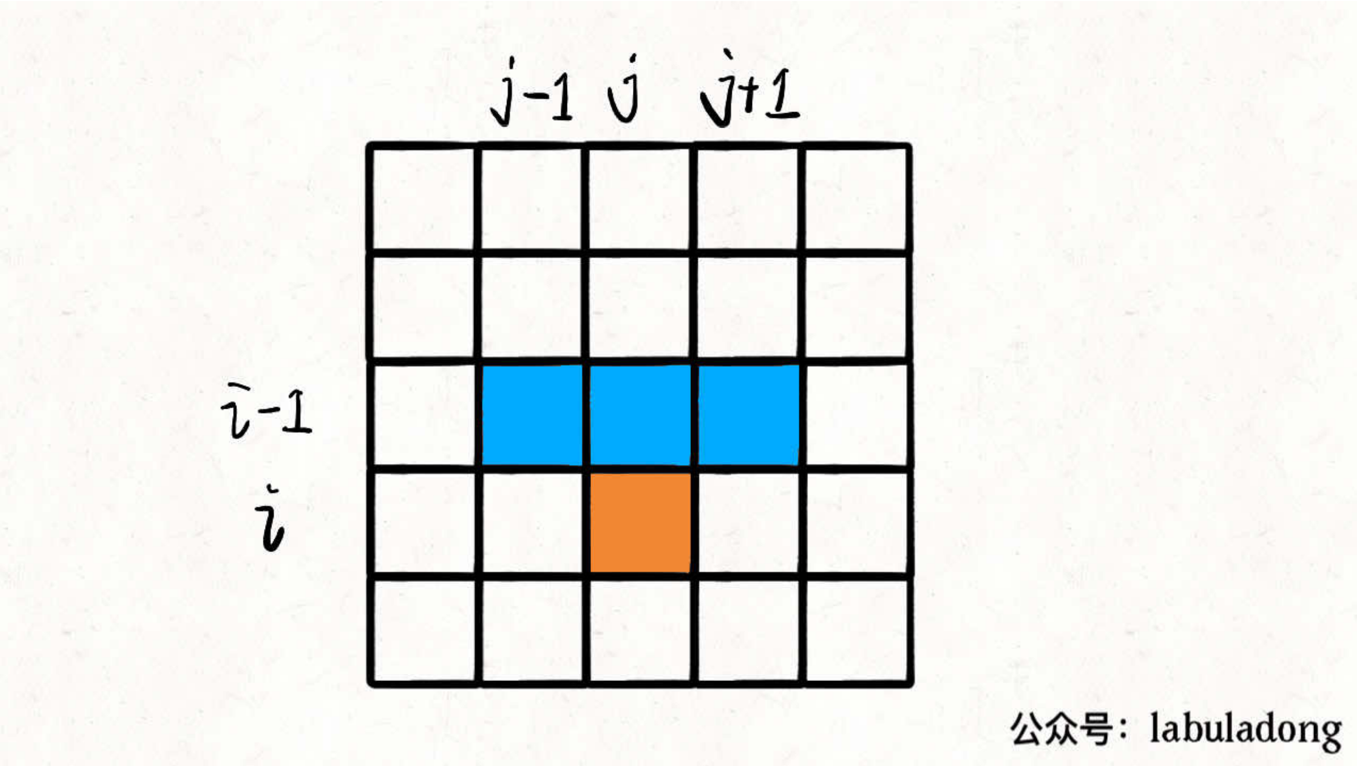
    // 终点可能在最后一行的任意一列
    for (int j = 0; j < n; j++) {
        res = Math.min(res, dp(matrix, n - 1, j));
    }

    return res;
}
```

因为我们可能落到最后一行的任意一列，所以要穷举一下，看看落到哪一列才能得到最小的路径和。

接下来看看 `dp` 函数如何实现。

对于 `matrix[i][j]`，只有可能从 `matrix[i-1][j]`, `matrix[i-1][j-1]`, `matrix[i-1][j+1]` 这三个位置转移过来。



那么，只要知道到达 `(i-1, j)`, `(i-1, j-1)`, `(i-1, j+1)` 这三个位置的最小路径和，加上 `matrix[i][j]` 的值，就能够计算出来到达位置 `(i, j)` 的最小路径和：

```
int dp(int[][] matrix, int i, int j) {
    // 非法索引检查
    if (i < 0 || j < 0 ||
        i >= matrix.length ||
        j >= matrix[0].length) {
        // 返回一个特殊值
        return 99999;
    }
    // base case
    if (i == 0) {
        return matrix[i][j];
    }
    // 状态转移
    return matrix[i][j] + min(
            dp(matrix, i - 1, j),
            dp(matrix, i - 1, j - 1),
            dp(matrix, i - 1, j + 1)
        );
}

int min(int a, int b, int c) {
    return Math.min(a, Math.min(b, c));
}
```

当然，上述代码是暴力穷举解法，我们可以用备忘录的方法消除重叠子问题，完整代码如下：

```
int minFallingPathSum(int[][] matrix) {
    int n = matrix.length;
    int res = Integer.MAX_VALUE;
    // 备忘录里的值初始化为 66666
    memo = new int[n][n];
    for (int i = 0; i < n; i++) {
        Arrays.fill(memo[i], 66666);
    }
    // 终点可能在 matrix[n-1] 的任意一列
    for (int j = 0; j < n; j++) {
        res = Math.min(res, dp(matrix, n - 1, j));
    }
    return res;
}

// 备忘录
int[][] memo;

int dp(int[][] matrix, int i, int j) {
    // 1、索引合法性检查
    if (i < 0 || j < 0 ||
        i >= matrix.length ||
        j >= matrix[0].length) {

        return 99999;
    }
    // 2、base case
    if (i == 0) {
        return matrix[0][j];
    }
    // 3、查找备忘录，防止重复计算
    if (memo[i][j] != 66666) {
        return memo[i][j];
    }
    // 进行状态转移
    memo[i][j] = matrix[i][j] + min(
            dp(matrix, i - 1, j),
            dp(matrix, i - 1, j - 1),
            dp(matrix, i - 1, j + 1)
        );
    return memo[i][j];
}

int min(int a, int b, int c) {
    return Math.min(a, Math.min(b, c));
}
```

如果看过我们公众号之前的动态规划系列文章，这个解题思路应该是非常容易理解的。

那么本文对于这个 dp 函数仔细探讨三个问题：

1、对于索引的合法性检测，返回值为什么是 99999？其他的值行不行？

2、base case 为什么是 `i == 0`？

3、备忘录 `memo` 的初始值为什么是 66666？其他值行不行？

首先，说说 base case 为什么是 `i == 0`，返回值为什么是 `matrix[0][j]`，这是根据 `dp` 函数的定义所决定的。

回顾我们的 `dp` 函数定义：

从第一行（`matrix[0][..]`）向下落，落到位置 `matrix[i][j]` 的最小路径和为 `dp(matrix, i, j)`。

根据这个定义，我们就是从 `matrix[0][j]` 开始下落。那如果我们想落到的目的地就是 `i == 0`，所需的路径和当然就是 `matrix[0][j]` 呗。

再说说备忘录 `memo` 的初始值为什么是 66666，这是由题目给出的数据范围决定的。

备忘录 `memo` 数组的作用是什么？

就是防止重复计算，将 `dp(matrix, i, j)` 的计算结果存进 `memo[i][j]`，遇到重复计算可以直接返回。

那么，我们必须要知道 `memo[i][j]` 到底存储计算结果没有，对吧？如果存结果了，就直接返回；没存，就去递归计算。

所以，`memo` 的初始值一定得是特殊值，和合法的答案有所区分。

我们回过头看看题目给出的数据范围：

> `matrix` 是 `n x n` 的二维数组，其中 `1 <= n <= 100`；对于二维数组中的元素，有 `-100 <= matrix[i][j] <= 100`。

假设 `matrix` 的大小是 100 x 100，所有元素都是 100，那么从第一行往下落，得到的路径和就是 100 x 100 = 10000，也就是最大的合法答案。

类似的，依然假设 `matrix` 的大小是 100 x 100，所有元素是 -100，那么从第一行往下落，就得到了最小的合法答案 -100 x 100 = -10000。

也就是说，这个问题的合法结果会落在区间 `[-10000, 10000]` 中。

所以，我们 `memo` 的初始值就要避开区间 `[-10000, 10000]`，换句话说，`memo` 的初始值只要在区间 `(-inf, -10001] U [10001, +inf)` 中就可以。

最后，说说对于不合法的索引，返回值应该如何确定，这需要根据我们状态转移方程的逻辑确定。

对于这道题，状态转移的基本逻辑如下：

```
int dp(int[][] matrix, int i, int j) {

    return matrix[i][j] + min(
            dp(matrix, i - 1, j),
            dp(matrix, i - 1, j - 1),
```

```
            dp(matrix, i - 1, j + 1)
        );
}
```

显然，`i - 1, j - 1, j + 1` 这几个运算可能会造成索引越界，对于索引越界的 `dp` 函数，应该返回一个不可能被取到的值。

因为我们调用的是 `min` 函数，最终返回的值是最小值，所以对于不合法的索引，只要 `dp` 函数返回一个永远不会被取到的最大值即可。

刚才说了，合法答案的区间是 `[-10000, 10000]`，所以我们的返回值只要大于 10000 就相当于一个永不会取到的最大值。

换句话说，只要返回区间 `[10001, +inf)` 中的一个值，就能保证不会被取到。

至此，我们就把动态规划相关的三个细节问题举例说明了。

拓展延伸一下，建议大家做题时，除了题意本身，一定不要忽视题目给定的其他信息。

本文举的例子，测试用例数据范围可以确定「什么是特殊值」，从而帮助我们将思路转化成代码。

除此之外，数据范围还可以帮我们估算算法的时间/空间复杂度。

比如说，有的算法题，你只想到一个暴力求解思路，时间复杂度比较高。如果发现题目给定的数据量比较大，那么肯定可以说明这个求解思路有问题或者存在优化的空间。

除了数据范围，有时候题目还会限制我们算法的时间复杂度，这种信息其实也暗示着一些东西。

比如要求我们的算法复杂度是 `O(NlogN)`，你想想怎么才能搞出一个对数级别的复杂度呢？

肯定得用到 二分搜索 或者二叉树相关的数据结构，比如 `TreeMap`，`PriorityQueue` 之类的对吧。

再比如，有时候题目要求你的算法时间复杂度是 `O(MN)`，这可以联想到什么？

可以大胆猜测，常规解法是用 回溯算法 暴力穷举，但是更好的解法是动态规划，而且是一个二维动态规划，需要一个 `M * N` 的二维 `dp` 数组，所以产生了这样一个时间复杂度。

如果你早就胸有成竹了，那就当我没说，毕竟猜测也不一定准确；但如果你本来就没啥解题思路，那有了这些推测之后，最起码可以给你的思路一些方向吧？

总之，多动脑筋，不放过任何蛛丝马迹，你不成为刷题小能手才怪。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

最优子结构和 dp 数组的遍历方向怎么定？

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

本文有视频版：动态规划详解进阶

这篇文章就给你讲明白三个问题：

1、到底什么才叫「最优子结构」，和动态规划什么关系。

2、如何判断一个问题是动态规划问题，即如何看出是否存在重叠子问题。

3、为什么动态规划遍历 `dp` 数组的方式五花八门，有的正着遍历，有的倒着遍历，有的斜着遍历。

一、最优子结构详解

「最优子结构」是某些问题的一种特定性质，并不是动态规划问题专有的。也就是说，很多问题其实都具有最优子结构，只是其中大部分不具有重叠子问题，所以我们不把它们归为动态规划系列问题而已。

我先举个很容易理解的例子：假设你们学校有 10 个班，你已经计算出了每个班的最高考试成绩。那么现在我要求你计算全校最高的成绩，你会不会算？当然会，而且你不用重新遍历全校学生的分数进行比较，而是只要在这 10 个最高成绩中取最大的就是全校的最高成绩。

我给你提出的这个问题就**符合最优子结构**：可以从子问题的最优结果推出更大规模问题的最优结果。让你算**每个班**的最优成绩就是子问题，你知道所有子问题的答案后，就可以借此推出**全校**学生的最优成绩这个规模更大的问题的答案。

你看，这么简单的问题都有最优子结构性质，只是因为显然没有重叠子问题，所以我们简单地求最值肯定用不出动态规划。

再举个例子：假设你们学校有 10 个班，你已知每个班的最大分数差（最高分和最低分的差值）。那么现在我让你计算全校学生中的最大分数差，你会不会算？可以想办法算，但是肯定不能通过已知的这 10 个班的最大分数差推到出来。因为这 10 个班的最大分数差不一定就包含全校学生的最大分数差，比如全校的最大分数差可能是 3 班的最高分和 6 班的最低分之差。

这次我给你提出的问题就**不符合最优子结构**，因为你没办通过每个班的最优值推出全校的最优值，没办法通过子问题的最优值推出规模更大的问题的最优值。前文 动态规划详解 说过，想满足最优子结，子问题之间必须互相独立。全校的最大分数差可能出现在两个班之间，显然子问题不独立，所以这个问题本身不符合最优子结构。

那么遇到这种最优子结构失效情况，怎么办？策略是：改造问题。对于最大分数差这个问题，我们不是没办法利用已知的每个班的分数差吗，那我只能这样写一段暴力代码：

```
int result = 0;
for (Student a : school) {
    for (Student b : school) {
        if (a is b) continue;
        result = max(result, |a.score - b.score|);
    }
}
return result;
```

改造问题，也就是把问题等价转化：最大分数差，不就等价于最高分数和最低分数的差么，那不就是要求最高和最低分数么，不就是我们讨论的第一个问题么，不就具有最优子结构了么？那现在改变思路，借助最优子结构解决最值问题，再回过头解决最大分数差问题，是不是就高效多了？

当然，上面这个例子太简单了，不过请读者回顾一下，我们做动态规划问题，是不是一直在求各种最值，本质跟我们举的例子没啥区别，无非需要处理一下重叠子问题。

前文不 同定义不同解法 和 高楼扔鸡蛋进阶 就展示了如何改造问题，不同的最优子结构，可能导致不同的解法和效率。

再举个常见但也十分简单的例子，求一棵二叉树的最大值，不难吧（简单起见，假设节点中的值都是非负数）：

```
int maxVal(TreeNode root) {
    if (root == null)
        return -1;
    int left = maxVal(root.left);
    int right = maxVal(root.right);
    return max(root.val, left, right);
}
```

你看这个问题也符合最优子结构，以 `root` 为根的树的最大值，可以通过两边子树（子问题）的最大值推导出来，结合刚才学校和班级的例子，很容易理解吧。

当然这也不是动态规划问题，旨在说明，最优子结构并不是动态规划独有的一种性质，能求最值的问题大部分都具有这个性质；**但反过来，最优子结构性质作为动态规划问题的必要条件，一定是让你求最值的**，以后碰到那种恶心人的最值题，思路往动态规划想就对了，这就是套路。

动态规划不就是从最简单的 base case 往后推导吗，可以想象成一个链式反应，以小博大。但只有符合最优子结构的问题，才有发生这种链式反应的性质。

找最优子结构的过程，其实就是证明状态转移方程正确性的过程，方程符合最优子结构就可以写暴力解了，写出暴力解就可以看出有没有重叠子问题了，有则优化，无则 OK。这也是套路，经常刷题的朋友应该能体会。

这里就不举那些正宗动态规划的例子了，读者可以翻翻历史文章，看看状态转移是如何遵循最优子结构的，这个话题就聊到这，下面再来看其他的动态规划迷惑行为。

二、如何一眼看出重叠子问题

经常有读者说：

看了前文 动态规划核心套路，我知道了如何一步步优化动态规划问题；

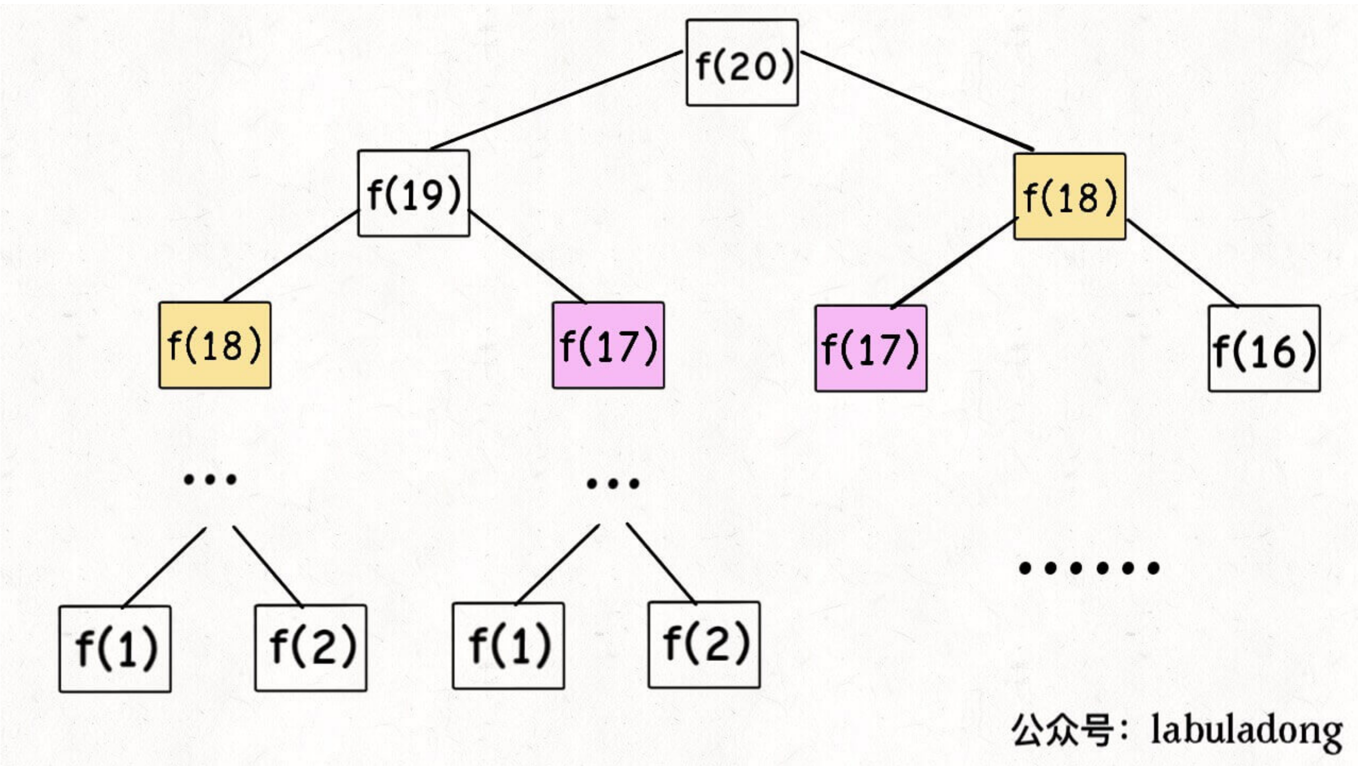
看了前文 动态规划设计：数学归纳法，我知道了利用数学归纳法写出暴力解（状态转移方程）。

但就算我写出了暴力解，我很难判断这个解法是否存在重叠子问题，从而无法确定是否可以运用备忘录等方法去优化算法效率。

对于这个问题，其实我在动态规划系列的文章中写过几次的，还是在这里再统一总结一下吧。

首先，最简单粗暴的方式就是画图，把递归树画出来，看看有没有重复的节点。

比如最简单的例子，动态规划核心套路 中斐波那契数列的递归树：



这棵递归树很明显存在重复的节点，所以我们可以通过备忘录避免冗余计算。

但毕竟斐波那契数列问题太简单了，实际的动态规划问题比较复杂，比如二维甚至三维的动态规划，当然也可以画递归树，但不免有些复杂。

比如在 最小路径和问题 中，我们写出了这样一个暴力解法：

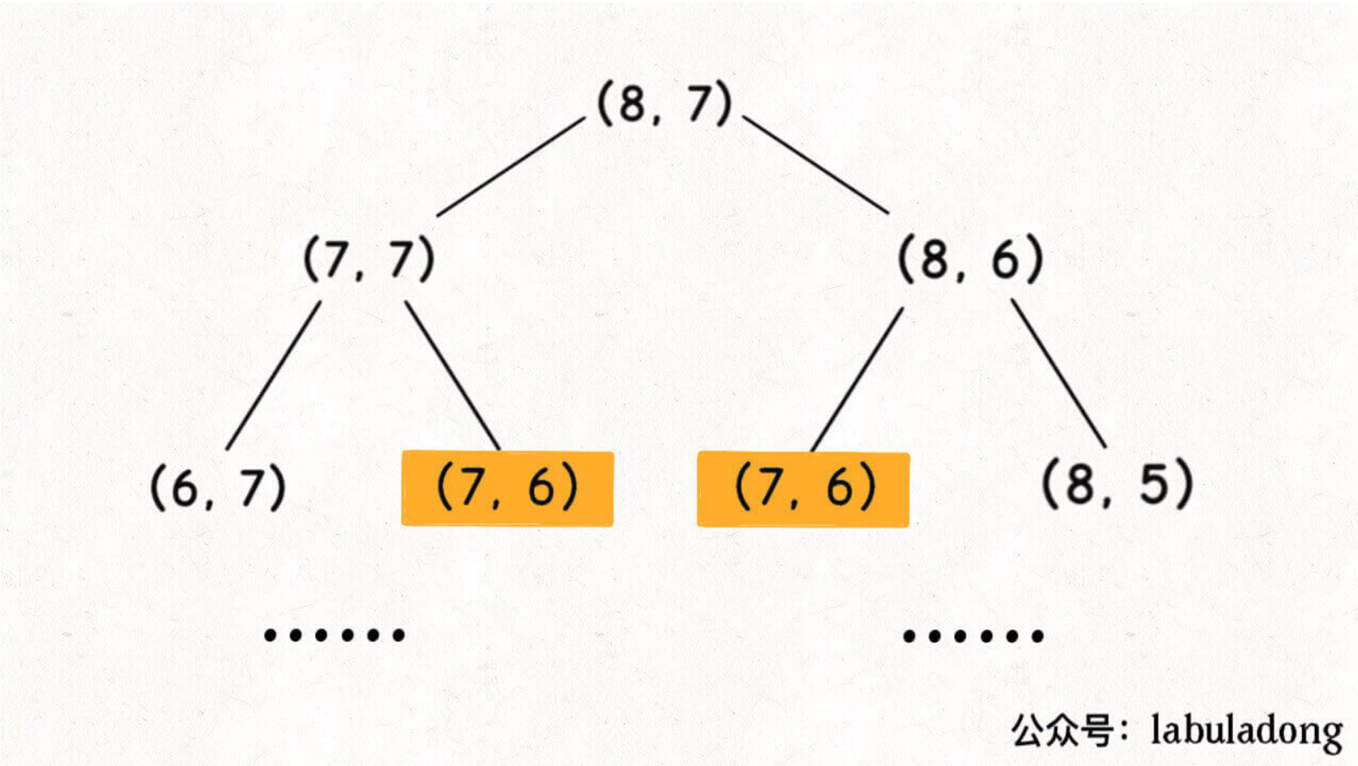
```
int dp(int[][] grid, int i, int j) {
    if (i == 0 && j == 0) {
        return grid[0][0];
    }
    if (i < 0 || j < 0) {
```

```
            return Integer.MAX_VALUE;
        }

        return Math.min(
                dp(grid, i - 1, j),
                dp(grid, i, j - 1)
            ) + grid[i][j];
    }
```

你不需要读过前文，光看这个函数代码就能看出来，该函数递归过程中参数 i, j 在不断变化，即「状态」是 (i, j) 的值，你是否可以判断这个解法是否存在重叠子问题呢？

假设输入的 i = 8, j = 7，二维状态的递归树如下图，显然出现了重叠子问题：



但稍加思考就可以知道，其实根本没必要画图，可以通过递归框架直接判断是否存在重叠子问题。

具体操作就是直接删掉代码细节，抽象出该解法的递归框架：

```
int dp(int[][] grid, int i, int j) {
    dp(grid, i - 1, j), // #1
    dp(grid, i, j - 1)  // #2
}
```

可以看到 i, j 的值在不断减小，那么我问你一个问题：如果我想从状态 (i, j) 转移到 (i-1, j-1)，有几种路径？

显然有两种路径，可以是 (i, j) -> #1 -> #2 或者 (i, j) -> #2 -> #1，不止一种，说明 (i-1, j-1) 会被多次计算，所以一定存在重叠子问题。

再举个稍微复杂的例子，前文 正则表达式问题 的暴力解代码：

```
bool dp(string& s, int i, string& p, int j) {
    int m = s.size(), n = p.size();
    if (j == n)  return i == m;
    if (i == m) {
        if ((n - j) % 2 == 1) return false;
        for (; j + 1 < n; j += 2) {
            if (p[j + 1] != '*') return false;
        }
        return true;
    }

    if (s[i] == p[j] || p[j] == '.') {
        if (j < n - 1 && p[j + 1] == '*') {
            return dp(s, i, p, j + 2)
               || dp(s, i + 1, p, j);
        } else {
            return dp(s, i + 1, p, j + 1);
        }
    } else if (j < n - 1 && p[j + 1] == '*') {
        return dp(s, i, p, j + 2);
    }
    return false;
}
```

代码有些复杂对吧，如果画图的话有些麻烦，但我们不画图，直接忽略所有细节代码和条件分支，只抽象出递归框架：

```
bool dp(string& s, int i, string& p, int j) {
    dp(s, i, p, j + 2);     // 1
    dp(s, i + 1, p, j);     // 2
    dp(s, i + 1, p, j + 1); // 3
    dp(s, i, p, j + 2);     // 4
}
```

和上一题一样，这个解法的「状态」也是 (i, j) 的值，那么我继续问你问题：如果我想从状态 (i, j) 转移到 (i+2, j+2)，有几种路径？

显然，至少有两条路径：(i, j) -> #1 -> #4 和 (i, j) -> #3 -> #3。

所以，不用画图就知道这个解法也存在重叠子问题，需要用备忘录技巧去优化。

三、dp 数组的遍历方向

我相信读者做动态规问题时，肯定会对 dp 数组的遍历顺序有些头疼。我们拿二维 dp 数组来举例，有时候我们是正向遍历：

```java
int[][] dp = new int[m][n];
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        // 计算 dp[i][j]
```

有时候我们反向遍历：

```java
for (int i = m - 1; i >= 0; i--)
    for (int j = n - 1; j >= 0; j--)
        // 计算 dp[i][j]
```

有时候可能会斜向遍历：

```java
// 斜着遍历数组
for (int l = 2; l <= n; l++) {
    for (int i = 0; i <= n - l; i++) {
        int j = l + i - 1;
        // 计算 dp[i][j]
    }
}
```

甚至更让人迷惑的是，有时候发现正向反向遍历都可以得到正确答案，比如我们在 团灭股票问题 中有的地方就正反皆可。

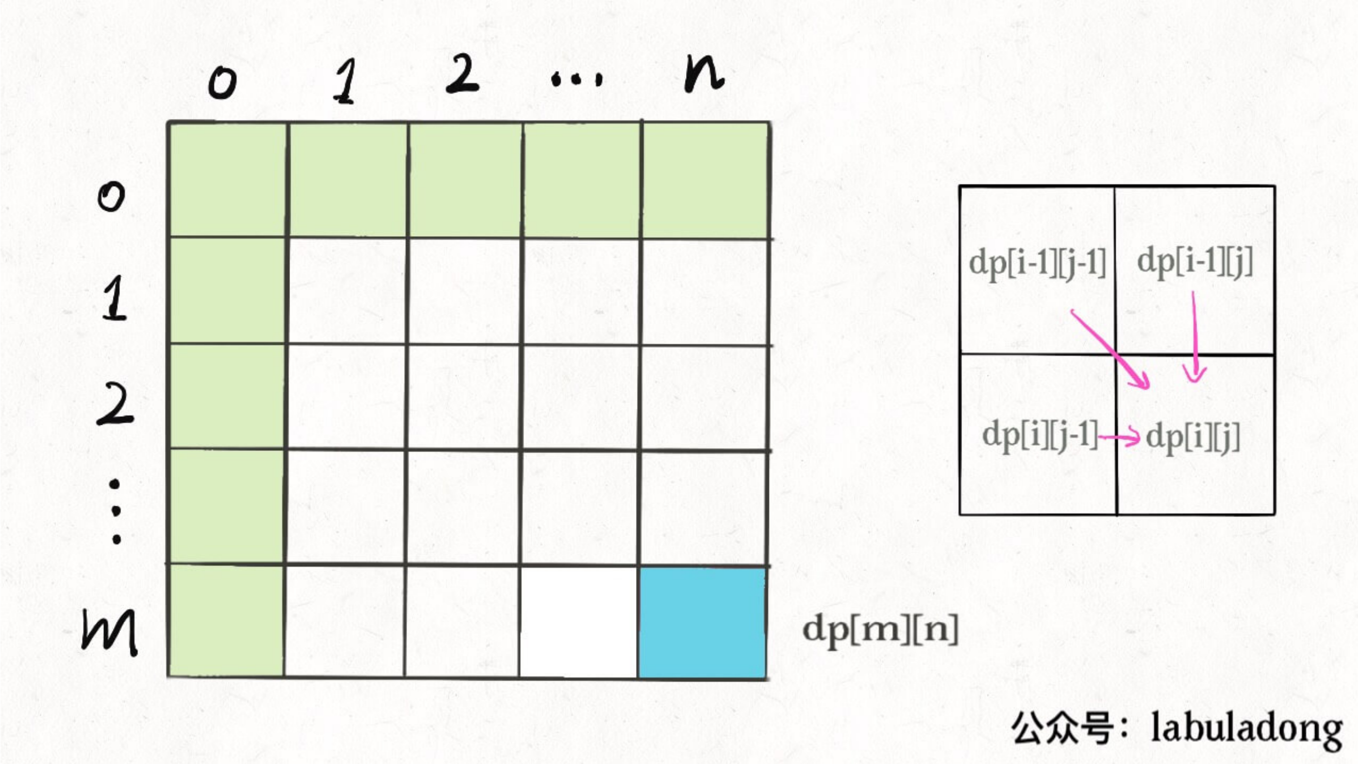
那么，如果仔细观察的话可以发现其中的原因的。你只要把住两点就行了：

1、遍历的过程中，所需的状态必须是已经计算出来的。

2、遍历结束后，存储结果的那个位置必须已经被计算出来。

下面来距离解释上面两个原则是什么意思。

比如编辑距离这个经典的问题，详解见前文 编辑距离详解，我们通过对 `dp` 数组的定义，确定了 base case 是 `dp[..][0]` 和 `dp[0][..]`，最终答案是 `dp[m][n]`；而且我们通过状态转移方程知道 `dp[i][j]` 需要从 `dp[i-1][j]`, `dp[i][j-1]`, `dp[i-1][j-1]` 转移而来，如下图：

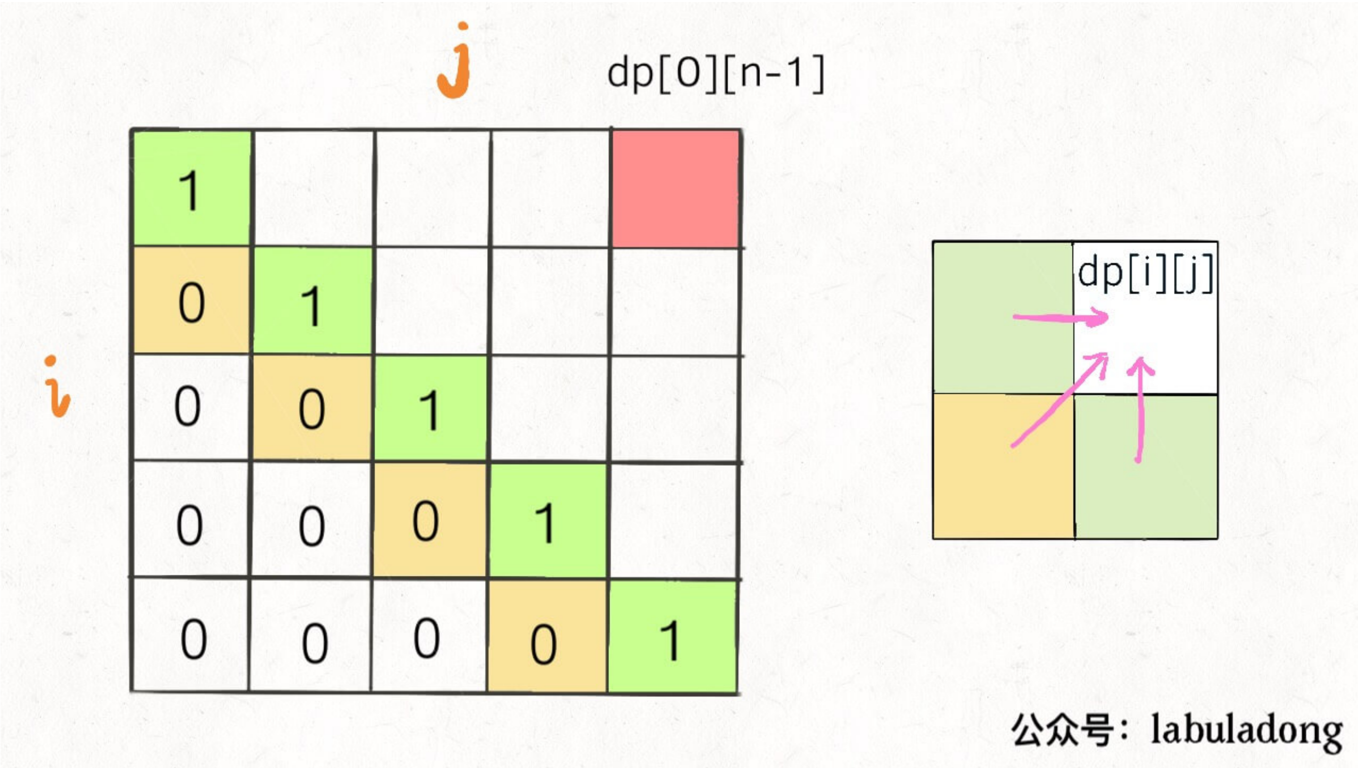


那么，参考刚才说的两条原则，你该怎么遍历 dp 数组？肯定是正向遍历：

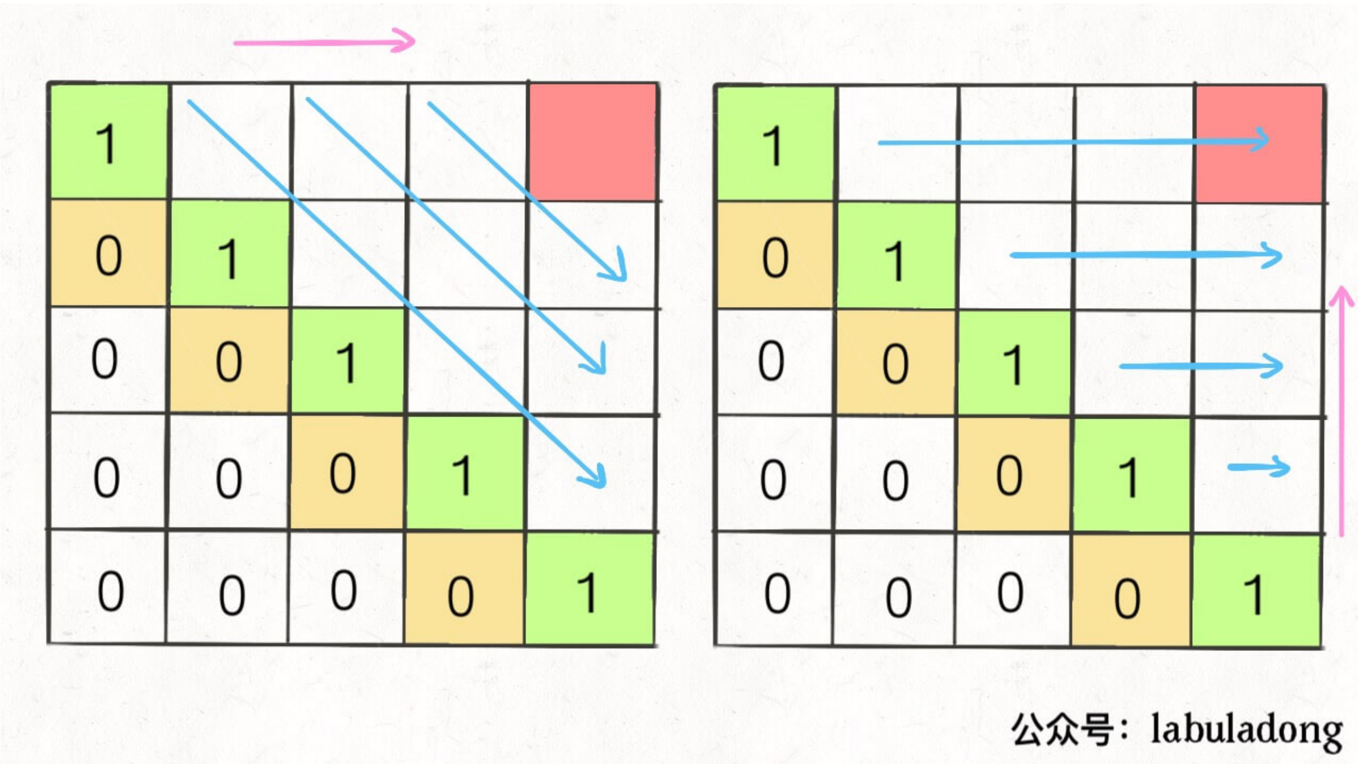
```
for (int i = 1; i < m; i++)
    for (int j = 1; j < n; j++)
        // 通过 dp[i-1][j], dp[i][j - 1], dp[i-1][j-1]
        // 计算 dp[i][j]
```

因为，这样每一步迭代的左边、上边、左上边的位置都是 base case 或者之前计算过的，而且最终结束在我们想要的答案 dp[m][n]。

再举一例，回文子序列问题，详见前文 子序列问题模板，我们通过过对 dp 数组的定义，确定了 base case 处在中间的对角线，dp[i][j] 需要从 dp[i+1][j], dp[i][j-1], dp[i+1][j-1] 转移而来，想要求的最终答案是 dp[0][n-1]，如下图：



这种情况根据刚才的两个原则，就可以有两种正确的遍历方式：



要么从左至右斜着遍历，要么从下向上从左到右遍历，这样才能保证每次 `dp[i][j]` 的左边、下边、左下边已经计算完毕，得到正确结果。

现在，你应该理解了这两个原则，主要就是看 base case 和最终结果的存储位置，保证遍历过程中使用的数据都是计算完毕的就行，有时候确实存在多种方法可以得到正确答案，可根据个人口味自行选择。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



提高刷题幸福感的小技巧

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 [点这里](#)。

相信每个人都有过被代码的小 bug 搞得心态爆炸的经历，本文分享一个我最常用的简单技巧，可以大幅提升刷题的幸福感。

在这之前，首先回答一个问题，刷力扣题是直接在网页上刷比较好还是在本地 IDE 上刷比较好？

如果是牛客网笔试那种自己处理输入输出的判题形式，一定要在 IDE 上写，这个没啥说的，但**像力扣这种判题形式，我个人偏好直接在网页上刷**，原因有二：

1、方便

因为力扣有的数据结构是自定的，比如说 `TreeNode`，`ListNode` 这种，在本地你还得把这个类 copy 过去。

而且在 IDE 上没办法测试，写完代码之后还得粘贴到网页上跑测试数据，那还不如直接网页上写呢。

算法又不是工程代码，量都比较小，IDE 的自动补全带来的收益基本可以忽略不计。

2、实用

到时候面试的时候，面试官给你出的算法题大都是希望你直接在网页上完成的，最好是边写边讲你的思路。

如果平时练习的时候就习惯没有 IDE 的自动补全，习惯手写代码大脑编译，到时候面试的时候写代码就能更快更从容。

之前我面快手的时候，有个面试官让我 [实现 LRU 算法](#)，我直接把双链表的实现、哈希链表的实现，在网页上全写出来了，而且一次无 bug 跑通，可以看到面试官惊讶的表情😂

我秋招能当 offer 收割机，很大程度上就是因为手写算法这一关超出面试官的预期，其实都是因为之前在网页上刷题练出来的。

接下来分享我觉得最常实用的干货技巧。

如何给算法 debug

代码的错误时无法避免的，有时候可能整个思路都错了，有时候可能是某些细节问题，比如 `i` 和 `j` 写反了，这种问题怎么排查？

我想一般的算法问题肯定不难排查，肉眼检查应该都没啥问题，再不济 `print` 打印一些关键变量的值，总能发现问题。

比较让人头疼的的应该是递归算法的问题排查。

如果没有一定的经验，函数递归的过程很难被正确理解，所以这里就重点讲讲如何高效 debug 递归算法。

有的读者可能会说，把算法 copy 到 IDE 里面，然后打断点一步步跟着走不就行了吗?

这个方法肯定是可以的，但是之前的文章多次说过，递归函数最好从一个全局的角度理解，而不要跳进具体的细节。

如果你对递归还不够熟悉，没有一个全局的视角，这种一步步打断点的方式也容易把人绕进去。

我的建议是直接在递归函数内部打印关键值，配合缩进，直观地观察递归函数执行情况。

最能提升我们 debug 效率的是缩进，除了解法函数，我们新定义一个函数 `printIndent` 和一个全局变量 `count`：

```
// 全局变量，记录递归函数的递归层数
int count = 0;

// 输入 n，打印 n 个 tab 缩进
void printIndent(int n) {
    for (int i = 0; i < n; i++) {
        printf("   ");
    }
}
```

接下来，套路来了：

在递归函数的开头，调用 `printIndent(count++)` 并打印关键变量；然后在所有 `return` 语句之前调用 `printIndent(--count)` 并打印返回值。

举个具体的例子，比如说上篇文章 练琴时悟出的一个动态规划算法 中实现了一个递归的 `dp` 函数，大致的结构如下：

```
int dp(string& ring, int i, string& key, int j) {
    /* base case */
    if (j == key.size()) {
        return 0;
    }

    /* 状态转移 */
    int res = INT_MAX;
    for (int k : charToIndex[key[j]]) {
        res = min(res, dp(ring, j, key, i + 1));
    }

    return res;
}
```

这个递归的 dp 函数在我进行了 debug 之后，变成了这样：

```
int count = 0;
void printIndent(int n) {
    for (int i = 0; i < n; i++) {
        printf("   ");
    }
}

int dp(string& ring, int i, string& key, int j) {
    // printIndent(count++);
    // printf("i = %d, j = %d\n", i, j);

    if (j == key.size()) {
        // printIndent(--count);
        // printf("return 0\n");
        return 0;
    }

    int res = INT_MAX;
    for (int k : charToIndex[key[j]]) {
        res = min(res, dp(ring, j, key, i + 1));
    }

    // printIndent(--count);
    // printf("return %d\n", res);
    return res;
}
```

就是在函数开头和所有 `return` 语句对应的地方加上一些打印代码。

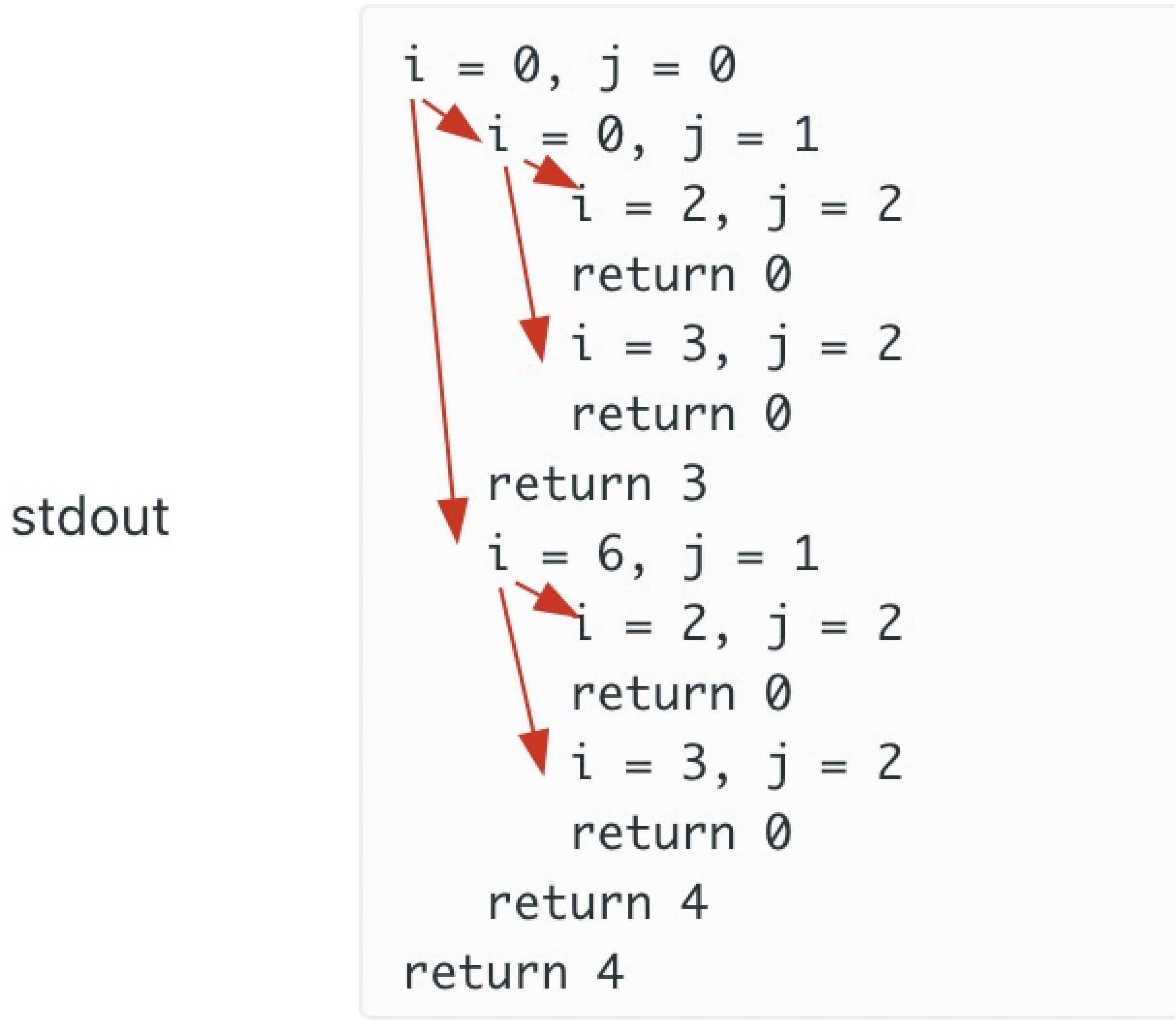
如果去掉注释，执行一个测试用例，输出如下：

stdout

```
i = 0, j = 0
    i = 0, j = 1
        i = 2, j = 2
        return 0
        i = 3, j = 2
        return 0
    return 3
    i = 6, j = 1
        i = 2, j = 2
        return 0
        i = 3, j = 2
        return 0
    return 4
return 4
```

这样，我们通过对比对应的缩进就能知道每次递归时输入的关键参数 `i, j` 的值，以及每次递归调用返回的结果是多少。

最重要的是，这样可以比较直观地看出递归过程，你有没有发现这就是一棵递归树？



前文 动态规划套路详解 说过，理解递归函数最重要的就是画出递归树，这样打印一下，连递归树都不用自己画了，而且还能清晰地看出每次递归的返回值。

可以说，这是对刷题「幸福感」提升最大的一个小技巧，比 IDE 打断点要高效。

好了，本文分享就到这里，马上快过年了，估计大家都无心学习了，不过刷题还是要坚持的，这就叫弯道超车，顺便实践一下这个技巧。

如果本文对你有帮助，点个在看，就会被推荐更多相似文章。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

4.2 经典动态规划



动态规划的底层逻辑也是穷举，只不过动态规划问题具有一些特殊的性质，使得穷举的过程中存在可优化的空间。

这里先提醒你，学习动态规划问题要格外注意这几个词：「**状态**」，「**选择**」，「**dp 数组的定义**」。你把这几个词理解到位了，就理解了动态规划的核心。

当然，动态规划问题的题型非常广泛，我不能保证你理解了核心就能做出所有动态规划题目，但我保证你理解了核心原理之后可以很轻松地理解别人的正确解法。如果自己勤加练习和总结，解决大部分中上难度的动态规划问题应该是没什么问题的。

公众号标签：手把手刷动态规划

最长递增子序列问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

300. 最长递增子序列（中等）

也许有读者看了前文 动态规划详解，学会了动态规划的套路：找到了问题的「状态」，明确了 dp 数组/函数的含义，定义了 base case；但是不知道如何确定「选择」，也就是找不到状态转移的关系，依然写不出动态规划解法，怎么办？

不要担心，动态规划的难点本来就在于寻找正确的状态转移方程，本文就借助经典的「最长递增子序列问题」来讲一讲设计动态规划的通用技巧：**数学归纳思想**。

最长递增子序列（Longest Increasing Subsequence，简写 LIS）是非常经典的一个算法问题，比较容易想到的是动态规划解法，时间复杂度 O(N^2)，我们借这个问题来由浅入深讲解如何找状态转移方程，如何写出动态规划解法。比较难想到的是利用二分查找，时间复杂度是 O(NlogN)，我们通过一种简单的纸牌游戏来辅助理解这种巧妙的解法。

力扣第 300 题「最长递增子序列」就是这个问题：

输入一个无序的整数数组，请你找到其中最长递增子序列的长度，函数签名如下：

```
int lengthOfLIS(int[] nums);
```

比如说输入 `nums=[10,9,2,5,3,7,101,18]`，其中最长的递增子序列是 `[2,3,7,101]`，所以算法的输出应该是 4。

注意「子序列」和「子串」这两个名词的区别，子串一定是连续的，而子序列不一定是连续的。下面先来设计动态规划算法解决这个问题。

一、动态规划解法

动态规划的核心设计思想是数学归纳法。

相信大家对数学归纳法都不陌生，高中就学过，而且思路很简单。比如我们想证明一个数学结论，那么**我们先假设这个结论在 `k < n` 时成立，然后根据这个假设，想办法推导证明出 `k = n` 的时候此结论也成立**。如果能够证明出来，那么就说明这个结论对于 `k` 等于任何数都成立。

类似的，我们设计动态规划算法，不是需要一个 dp 数组吗？我们可以假设 `dp[0...i-1]` 都已经被算出来了，然后问自己：怎么通过这些结果算出 `dp[i]`？

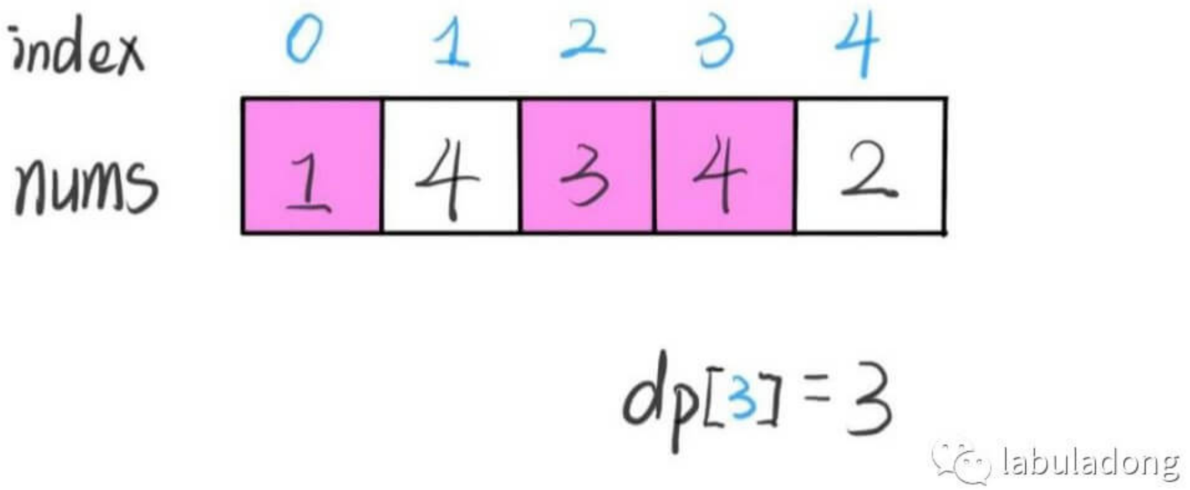
直接拿最长递增子序列这个问题举例你就明白了。不过，首先要定义清楚 dp 数组的含义，即 `dp[i]` 的值到底代表着什么？

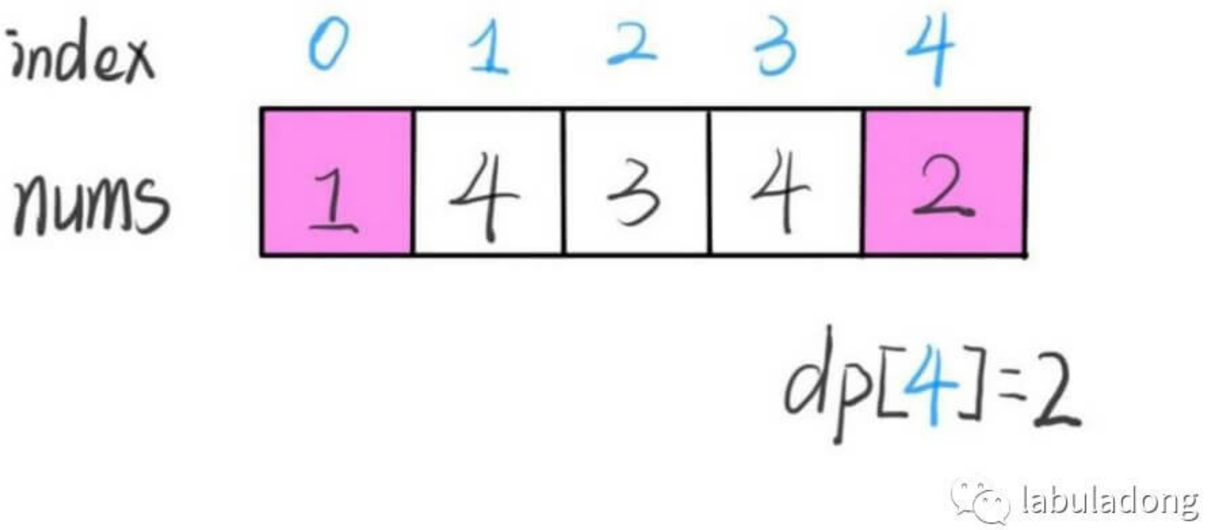
我们的定义是这样的：`dp[i]` 表示以 `nums[i]` 这个数结尾的最长递增子序列的长度。

> PS：为什么这样定义呢？这是解决子序列问题的一个套路，后文动态规划之子序列问题解题模板 总结了几种常见套路。你读完本章所有的动态规划问题，就会发现 `dp` 数组的定义方法也就那几种。

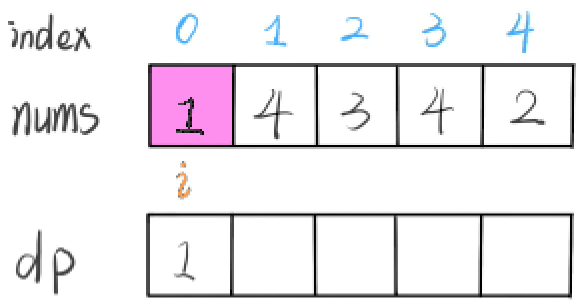
根据这个定义，我们就可以推出 base case：`dp[i]` 初始值为 1，因为以 `nums[i]` 结尾的最长递增子序列起码要包含它自己。

举两个例子：





算法演进的过程是这样的，：



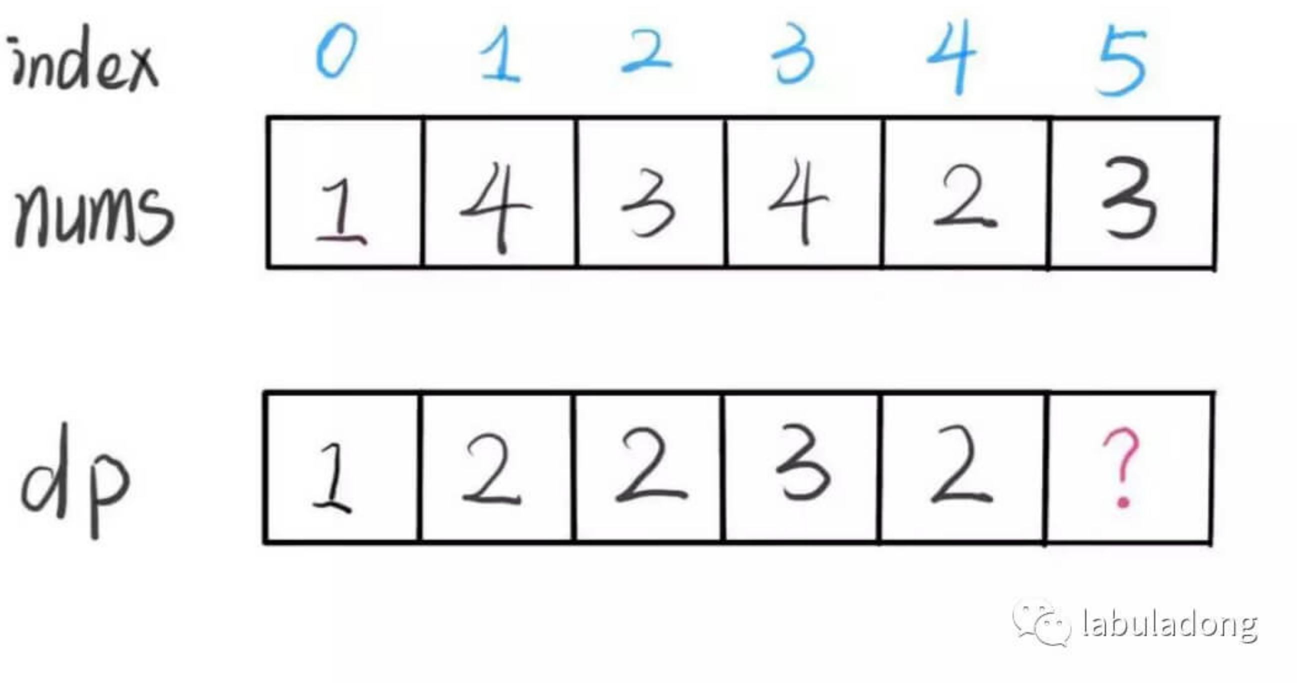
根据这个定义，我们的最终结果（子序列的最大长度）应该是 dp 数组中的最大值。

```
int res = 0;
for (int i = 0; i < dp.length; i++) {
    res = Math.max(res, dp[i]);
}
return res;
```

读者也许会问，刚才的算法演进过程中每个 `dp[i]` 的结果是我们肉眼看出来的，我们应该怎么设计算法逻辑来正确计算每个 `dp[i]` 呢？

这就是动态规划的重头戏了，要思考如何设计算法逻辑进行状态转移，才能正确运行呢？这里就可以使用数学归纳的思想：

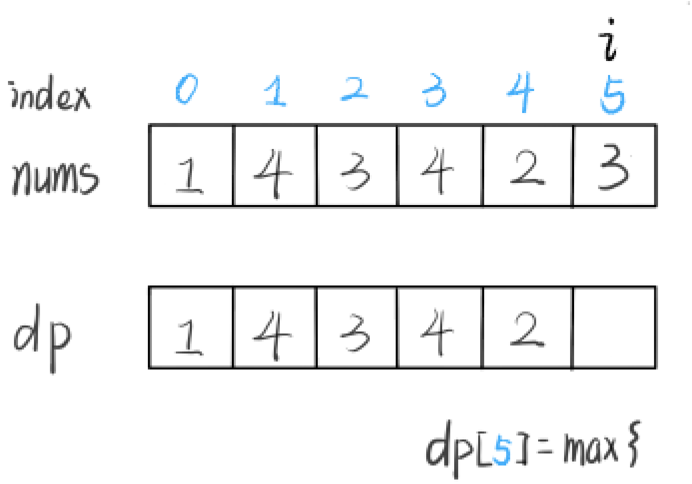
假设我们已经知道了 `dp[0..4]` 的所有结果，我们如何通过这些已知结果推出 `dp[5]` 呢？



根据刚才我们对 `dp` 数组的定义，现在想求 `dp[5]` 的值，也就是想求以 `nums[5]` 为结尾的最长递增子序列。

`nums[5] = 3`，既然是递增子序列，我们只要找到前面那些结尾比 3 小的子序列，然后把 3 接到最后，就可以形成一个新的递增子序列，而且这个新的子序列长度加一。

显然，可能形成很多种新的子序列，但是我们只选择最长的那一个，把最长子序列的长度作为 `dp[5]` 的值即可。




```
for (int j = 0; j < i; j++) {
    if (nums[i] > nums[j])
```

```
            dp[i] = Math.max(dp[i], dp[j] + 1);
    }
```

当 `i = 5` 时，这段代码的逻辑就可以算出 `dp[5]`。其实到这里，这道算法题我们就基本做完了。

读者也许会问，我们刚才只是算了 `dp[5]` 呀，`dp[4]`, `dp[3]` 这些怎么算呢？类似数学归纳法，你已经可以算出 `dp[5]` 了，其他的就都可以算出来：

```
for (int i = 0; i < nums.length; i++) {
    for (int j = 0; j < i; j++) {
        if (nums[i] > nums[j])
            dp[i] = Math.max(dp[i], dp[j] + 1);
    }
}
```

结合我们刚才说的 base case，下面我们看一下完整代码：

```
int lengthOfLIS(int[] nums) {
    // 定义：dp[i] 表示以 nums[i] 这个数结尾的最长递增子序列的长度
    int[] dp = new int[nums.length];
    // base case：dp 数组全都初始化为 1
    Arrays.fill(dp, 1);
    for (int i = 0; i < nums.length; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j])
                dp[i] = Math.max(dp[i], dp[j] + 1);
        }
    }

    int res = 0;
    for (int i = 0; i < dp.length; i++) {
        res = Math.max(res, dp[i]);
    }
    return res;
}
```

至此，这道题就解决了，时间复杂度 O(N^2)。总结一下如何找到动态规划的状态转移关系：

1、明确 `dp` 数组的定义。这一步对于任何动态规划问题都很重要，如果不得当或者不够清晰，会阻碍之后的步骤。

2、根据 `dp` 数组的定义，运用数学归纳法的思想，假设 `dp[0...i-1]` 都已知，想办法求出 `dp[i]`，一旦这一步完成，整个题目基本就解决了。

但如果无法完成这一步，很可能就是 `dp` 数组的定义不够恰当，需要重新定义 `dp` 数组的含义；或者可能是 `dp` 数组存储的信息还不够，不足以推出下一步的答案，需要把 `dp` 数组扩大成二维数组甚至三维数组。

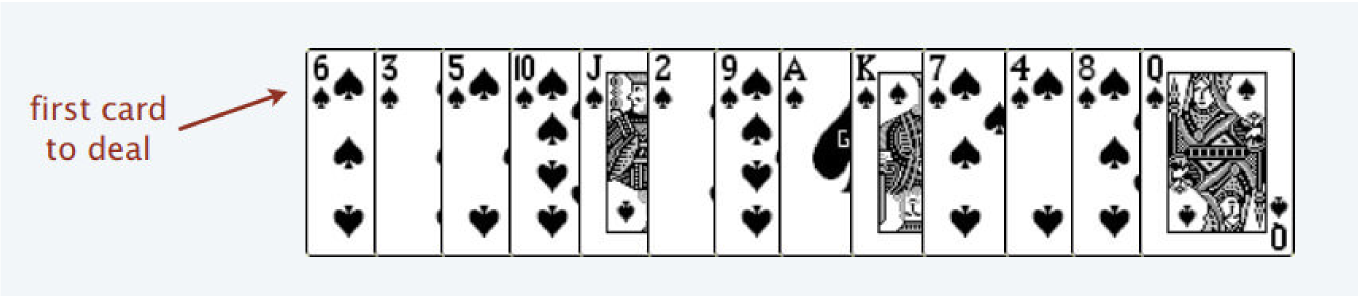
二、二分查找解法

这个解法的时间复杂度为 O(NlogN)，但是说实话，正常人基本想不到这种解法（也许玩过某些纸牌游戏的人可以想出来）。所以大家了解一下就好，正常情况下能够给出动态规划解法就已经很不错了。

根据题目的意思，我都很难想象这个问题竟然能和二分查找扯上关系。其实最长递增子序列和一种叫做 patience game 的纸牌游戏有关，甚至有一种排序方法就叫做 patience sorting（耐心排序）。

为了简单起见，后文跳过所有数学证明，通过一个简化的例子来理解一下算法思路。

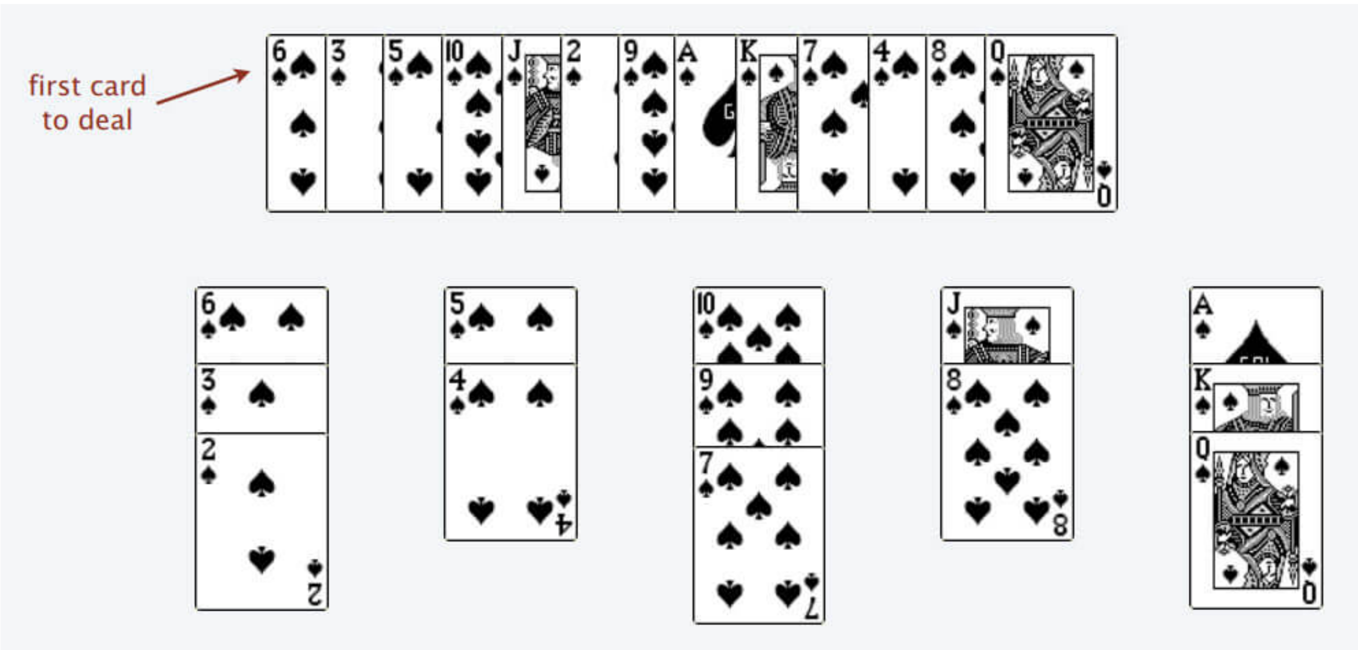
首先，给你一排扑克牌，我们像遍历数组那样从左到右一张一张处理这些扑克牌，最终要把这些牌分成若干堆。



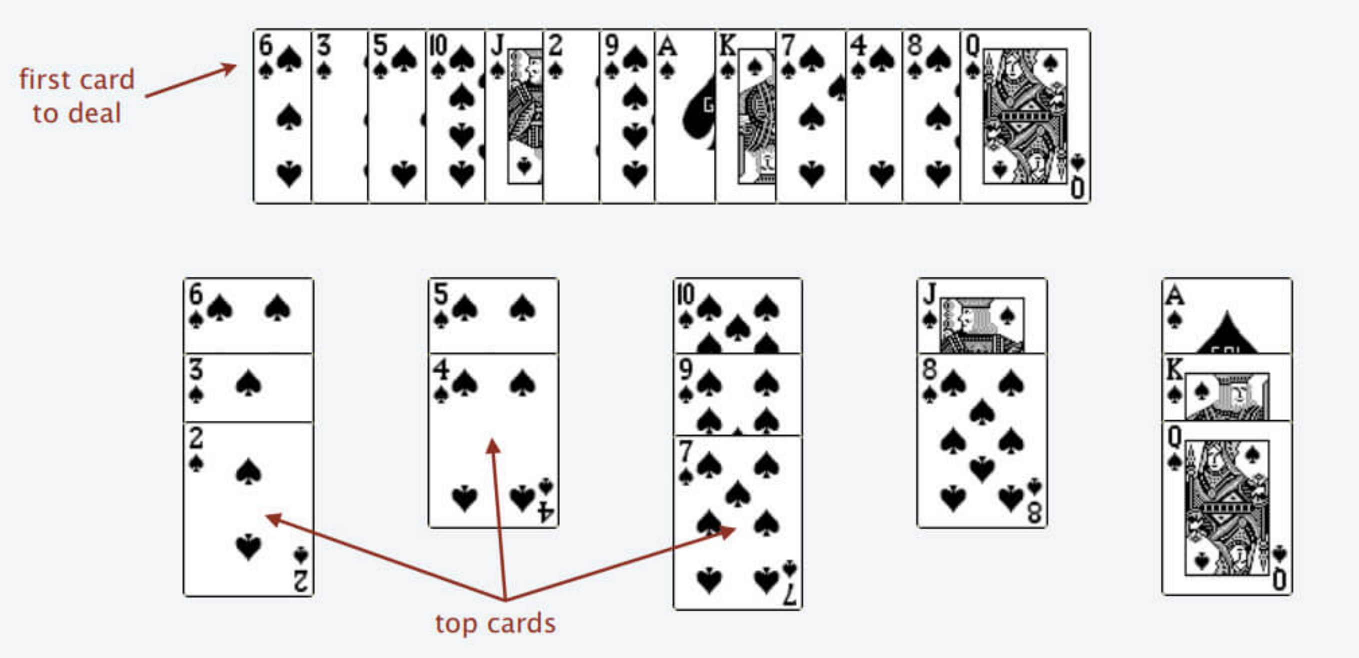
处理这些扑克牌要遵循以下规则：

只能把点数小的牌压到点数比它大的牌上；如果当前牌点数较大没有可以放置的堆，则新建一个堆，把这张牌放进去；如果当前牌有多个堆可供选择，则选择最左边的那一堆放置。

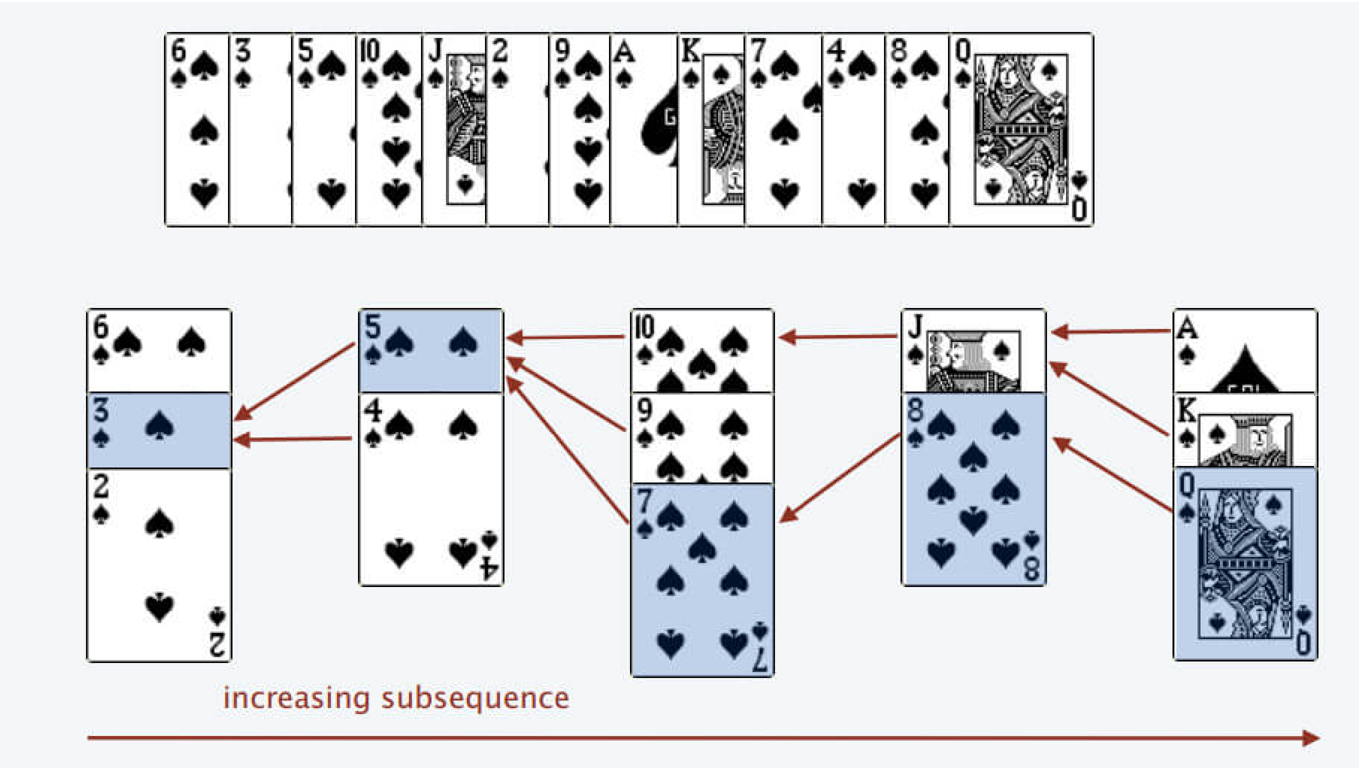
比如说上述的扑克牌最终会被分成这样 5 堆（我们认为纸牌 A 的牌面是最大的，纸牌 2 的牌面是最小的）。



为什么遇到多个可选择堆的时候要放到最左边的堆上呢？因为这样可以保证牌堆顶的牌有序（2, 4, 7, 8, Q），证明略。



按照上述规则执行，可以算出最长递增子序列，牌的堆数就是最长递增子序列的长度，证明略。



我们只要把处理扑克牌的过程编程写出来即可。每次处理一张扑克牌不是要找一个合适的牌堆顶来放吗，牌堆顶的牌不是**有序**吗，这就能用到二分查找了：用二分查找来搜索当前牌应放置的位置。

> PS：旧文[二分查找算法详解](#)详细介绍了二分查找的细节及变体，这里就完美应用上了，如果没读过强烈建议阅读。

```java
public int lengthOfLIS(int[] nums) {
    int[] top = new int[nums.length];
    // 牌堆数初始化为 0
    int piles = 0;
    for (int i = 0; i < nums.length; i++) {
```

```
        // 要处理的扑克牌
        int poker = nums[i];

        /***** 搜索左侧边界的二分查找 *****/
        int left = 0, right = piles;
        while (left < right) {
            int mid = (left + right) / 2;
            if (top[mid] > poker) {
                right = mid;
            } else if (top[mid] < poker) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        /*********************************/

        // 没找到合适的牌堆，新建一堆
        if (left == piles) piles++;
        // 把这张牌放到牌堆顶
        top[left] = poker;
    }
    // 牌堆数就是 LIS 长度
    return piles;
}
```

至此，二分查找的解法也讲解完毕。

这个解法确实很难想到。首先涉及数学证明，谁能想到按照这些规则执行，就能得到最长递增子序列呢？其次还有二分查找的运用，要是对二分查找的细节不清楚，给了思路也很难写对。

所以，这个方法作为思维拓展好了。但动态规划的设计方法应该完全理解：假设之前的答案已知，利用数学归纳的思想正确进行状态的推演转移，最终得到答案。

接下来可阅读：

- 动态规划之最大子数组

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

最大子数组和问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

53. 最大子序和（简单）

力扣第 53 题「最大子序和」问题和前文讲过的 经典动态规划：最长递增子序列 的套路非常相似，代表着一类比较特殊的动态规划问题的思路。

题目如下：

给你输入一个整数数组 `nums`，请你找在其中找一个和最大的子数组，返回这个子数组的和。

函数签名如下：

```
int maxSubArray(int[] nums);
```

比如说输入 `nums = [-3,1,3,-1,2,-4,2]`，算法返回 5，因为最大子数组 `[1,3,-1,2]` 的和为 5。

思路分析

其实第一次看到这道题，我首先想到的是滑动窗口算法，因为我们前文说过嘛，滑动窗口算法就是专门处理子串/子数组问题的，这里不就是子数组问题么？

但是，稍加分析就发现，**这道题还不能用滑动窗口算法，因为数组中的数字可以是负数。**

滑动窗口算法无非就是双指针形成的窗口扫描整个数组/子串，但关键是，你得清楚地知道什么时候应该移动右侧指针来扩大窗口，什么时候移动左侧指针来减小窗口。

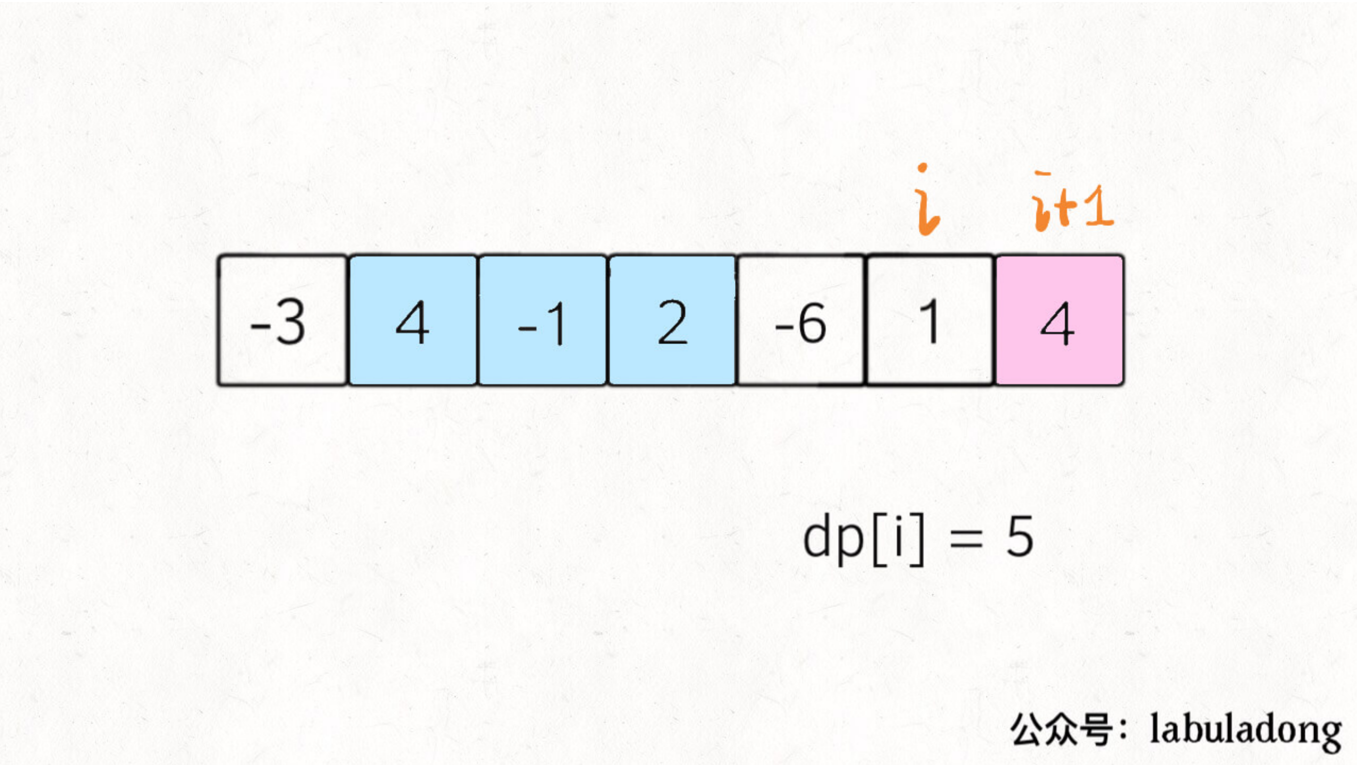
而对于这道题目，你想想，当窗口扩大的时候可能遇到负数，窗口中的值也就可能增加也可能减少，这种情况下不知道什么时机去收缩左侧窗口，也就无法求出「最大子数组和」。

解决这个问题需要动态规划技巧，但是 `dp` 数组的定义比较特殊。按照我们常规的动态规划思路，一般是这样定义 `dp` 数组：

`nums[0..i]` 中的「最大的子数组和」为 `dp[i]`。

如果这样定义的话，整个 nums 数组的「最大子数组和」就是 dp[n-1]。如何找状态转移方程呢？按照数学归纳法，假设我们知道了 dp[i-1]，如何推导出 dp[i] 呢？

如下图，按照我们刚才对 dp 数组的定义，dp[i] = 5，也就是等于 nums[0..i] 中的最大子数组和：



那么在上图这种情况中，利用数学归纳法，你能用 dp[i] 推出 dp[i+1] 吗？

实际上是不行的，因为子数组一定是连续的，按照我们当前 dp 数组定义，并不能保证 nums[0..i] 中的最大子数组与 nums[i+1] 是相邻的，也就没办法从 dp[i] 推导出 dp[i+1]。

所以说我们这样定义 dp 数组是不正确的，无法得到合适的状态转移方程。对于这类子数组问题，我们就要重新定义 dp 数组的含义：

以 nums[i] 为结尾的「最大子数组和」为 dp[i]。

这种定义之下，想得到整个 nums 数组的「最大子数组和」，不能直接返回 dp[n-1]，而需要遍历整个 dp 数组：

```
int res = Integer.MIN_VALUE;
for (int i = 0; i < n; i++) {
    res = Math.max(res, dp[i]);
}
return res;
```

依然使用数学归纳法来找状态转移关系：假设我们已经算出了 dp[i-1]，如何推导出 dp[i] 呢？

可以做到，dp[i] 有两种「选择」，要么与前面的相邻子数组连接，形成一个和更大的子数组；要么不与前面的子数组连接，自成一派，自己作为一个子数组。

如何选择？既然要求「最大子数组和」，当然选择结果更大的那个啦：

```
// 要么自成一派，要么和前面的子数组合并
dp[i] = Math.max(nums[i], nums[i] + dp[i - 1]);
```

综上，我们已经写出了状态转移方程，就可以直接写出解法了：

```
int maxSubArray(int[] nums) {
    int n = nums.length;
    if (n == 0) return 0;
    // 定义：dp[i] 记录以 nums[i] 为结尾的「最大子数组和」
    int[] dp = new int[n];
    // base case
    // 第一个元素前面没有子数组
    dp[0] = nums[0];
    // 状态转移方程
    for (int i = 1; i < n; i++) {
        dp[i] = Math.max(nums[i], nums[i] + dp[i - 1]);
    }
    // 得到 nums 的最大子数组
    int res = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        res = Math.max(res, dp[i]);
    }
    return res;
}
```

以上解法时间复杂度是 O(N)，空间复杂度也是 O(N)，较暴力解法已经很优秀了，不过**注意到 `dp[i]` 仅仅和 `dp[i-1]` 的状态有关**，那么我们可以施展前文 动态规划的降维打击 讲的技巧进行进一步优化，将空间复杂度降低：

```
int maxSubArray(int[] nums) {
    int n = nums.length;
    if (n == 0) return 0;
    // base case
    int dp_0 = nums[0];
    int dp_1 = 0, res = dp_0;

    for (int i = 1; i < n; i++) {
        // dp[i] = max(nums[i], nums[i] + dp[i-1])
        dp_1 = Math.max(nums[i], nums[i] + dp_0);
        dp_0 = dp_1;
        // 顺便计算最大的结果
        res = Math.max(res, dp_1);
    }

    return res;
}
```

最后总结

虽然说动态规划推状态转移方程确实比较玄学，但大部分还是有些规律可循的。

今天这道「最大子数组和」就和「最长递增子序列」非常类似，`dp` 数组的定义是「以 `nums[i]` 为结尾的最大子数组和/最长递增子序列为 `dp[i]`」。因为只有这样定义才能将 `dp[i+1]` 和 `dp[i]` 建立起联系，利用数学归纳法写出状态转移方程。

如果你有高质量的问题，可以 [点击这里](#) 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



最长公共子序列问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

1143. 最长公共子序列（中等）

583. 两个字符串的删除操作（中等）

712. 两个字符串的最小ASCII删除和（中等）

好久没写动态规划算法相关的文章了，今天来搞一把。

不知道大家做算法题有什么感觉，**我总结出来做算法题的技巧就是，把大的问题细化到一个点，先研究在这个小的点上如何解决问题，然后再通过递归/迭代的方式扩展到整个问题。**

比如说我们前文 手把手带你刷二叉树第三期，解决二叉树的题目，我们就会把整个问题细化到某一个节点上，想象自己站在某个节点上，需要做什么，然后套二叉树递归框架就行了。

动态规划系列问题也是一样，尤其是子序列相关的问题。**本文从「最长公共子序列问题」展开，总结三道子序列问题**，解这道题仔细讲讲这种子序列问题的套路，你就能感受到这种思维方式了。

最长公共子序列

计算最长公共子序列（Longest Common Subsequence，简称 LCS）是一道经典的动态规划题目，大家应该都见过：

给你输入两个字符串 `s1` 和 `s2`，请你找出他们俩的最长公共子序列，返回这个子序列的长度。

力扣第 1143 题就是这道题，函数签名如下：

```
int longestCommonSubsequence(String s1, String s2);
```

比如说输入 `s1 = "zabcde", s2 = "acez"`，它俩的最长公共子序列是 `lcs = "ace"`，长度为 3，所以算法返回 3。

如果没有做过这道题，一个最简单的暴力算法就是，把 `s1` 和 `s2` 的所有子序列都穷举出来，然后看看有没有公共的，然后在所有公共子序列里面再寻找一个长度最大的。

显然，这种思路的复杂度非常高，你要穷举出所有子序列，这个复杂度就是指数级的，肯定不实际。

正确的思路是不要考虑整个字符串，而是细化到 `s1` 和 `s2` 的每个字符。前文 子序列解题模板 中总结的一个规律：

应合作方要求，本文不便在此发布，请扫码关注回复关键词「LCS」查看：

编辑距离问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

72. 编辑距离（困难）

本文有视频版：编辑距离详解动态规划

前几天看了一份鹅厂的面试题，算法部分大半是动态规划，最后一题就是写一个计算编辑距离的函数，今天就专门写一篇文章来探讨一下这个问题。

我个人很喜欢编辑距离这个问题，因为它看起来十分困难，解法却出奇得简单漂亮，而且它是少有的比较实用的算法（我承认很多算法问题都不太实用）。

力扣第 72 题「编辑距离」就是这个问题，先看下题目：

72. 编辑距离 labuladong 题解 思路

难度 困难 2122

给你两个单词 `s1` 和 `s2`，请返回将 `s1` 转换成 `s2` 最少的操作数。

你可以对一个单词进行如下三种操作：

- 插入一个字符
- 删除一个字符
- 替换一个字符

示例 1：

```
输入：s1 = "horse", s2 = "ros"
输出：3
解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

示例 2：

```
输入：s1 = "intention", s2 = "execution"
输出：5
解释：
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')
```

函数签名如下：

```
def minDistance(s1: str, s2: str) -> int:
```

为什么说这个问题难呢，因为显而易见，它就是难，让人手足无措，望而生畏。

为什么说它实用呢，因为前几天我就在日常生活中用到了这个算法。之前有一篇公众号文章由于疏忽，写错位了一段内容，我决定修改这部分内容让逻辑通顺。但是公众号文章最多只能修改 20 个字，且只支持增、删、替换操作（跟编辑距离问题一模一样），于是我就用算法求出了一个最优方案，只用了 16 步就完成了修改。

再比如高大上一点的应用，DNA 序列是由 A,G,C,T 组成的序列，可以类比成字符串。编辑距离可以衡量两个 DNA 序列的相似度，编辑距离越小，说明这两段 DNA 越相似，说不定这俩 DNA 的主人是远古近亲啥的。

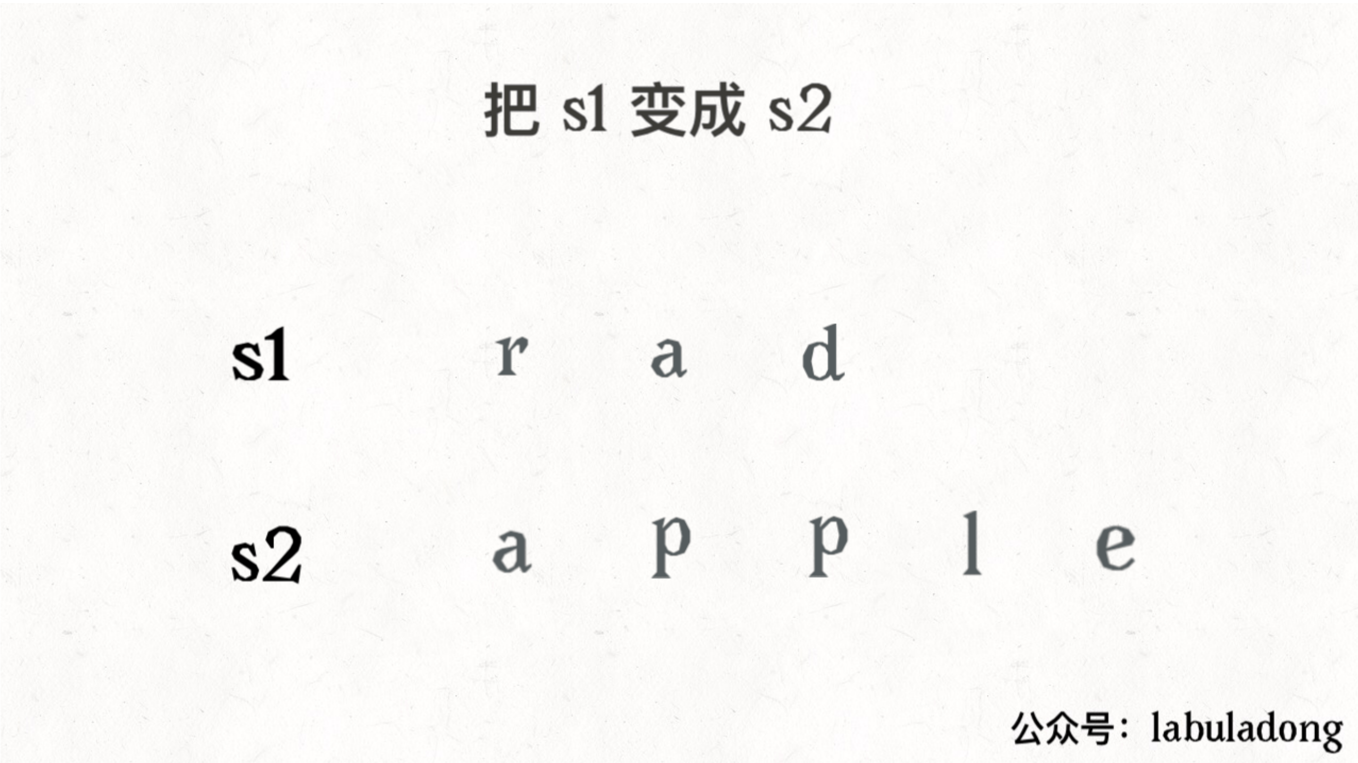
下面言归正传，详细讲解一下编辑距离该怎么算，相信本文会让你有收获。

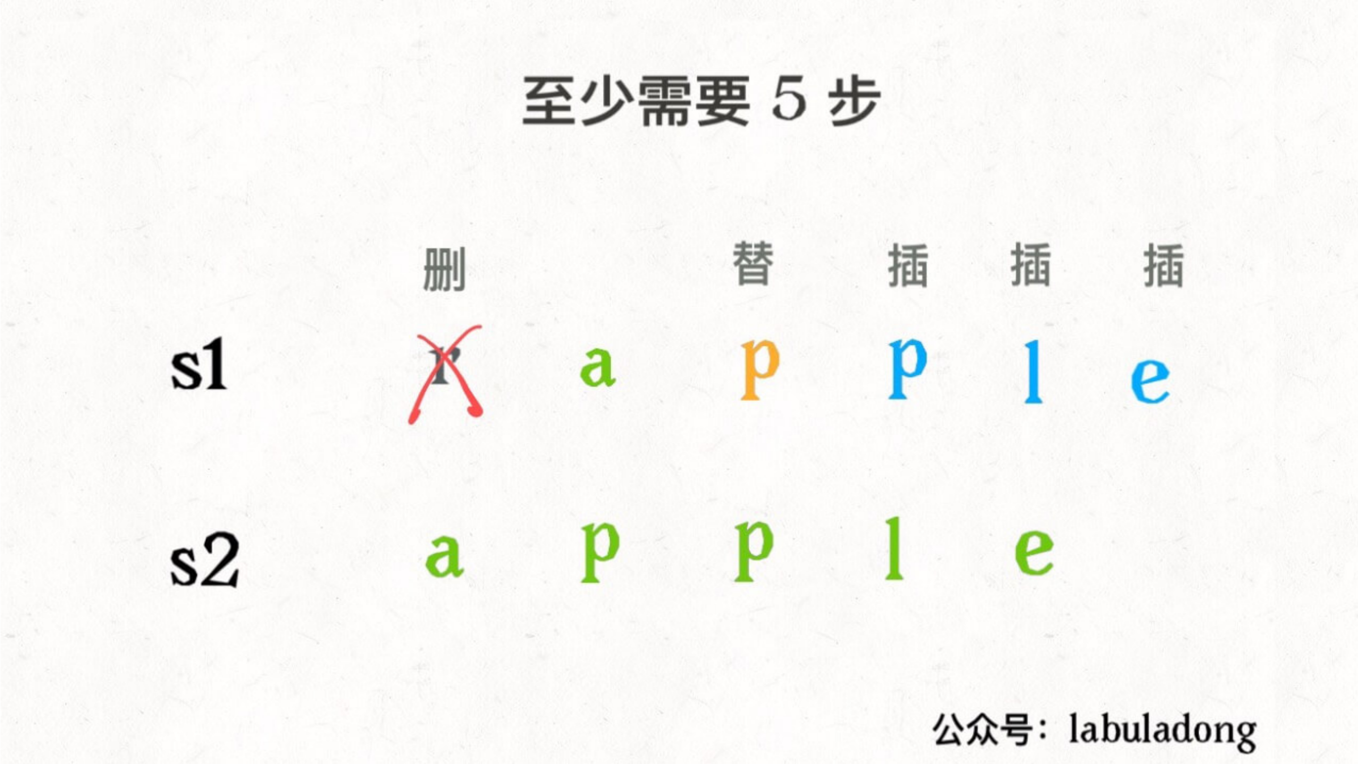
一、思路

编辑距离问题就是给我们两个字符串 `s1` 和 `s2`，只能用三种操作，让我们把 `s1` 变成 `s2`，求最少的操作数。需要明确的是，不管是把 `s1` 变成 `s2` 还是反过来，结果都是一样的，所以后文就以 `s1` 变成 `s2` 举例。

前文 最长公共子序列 说过，**解决两个字符串的动态规划问题，一般都是用两个指针 `i,j` 分别指向两个字符串的最后，然后一步步往前走，缩小问题的规模。**

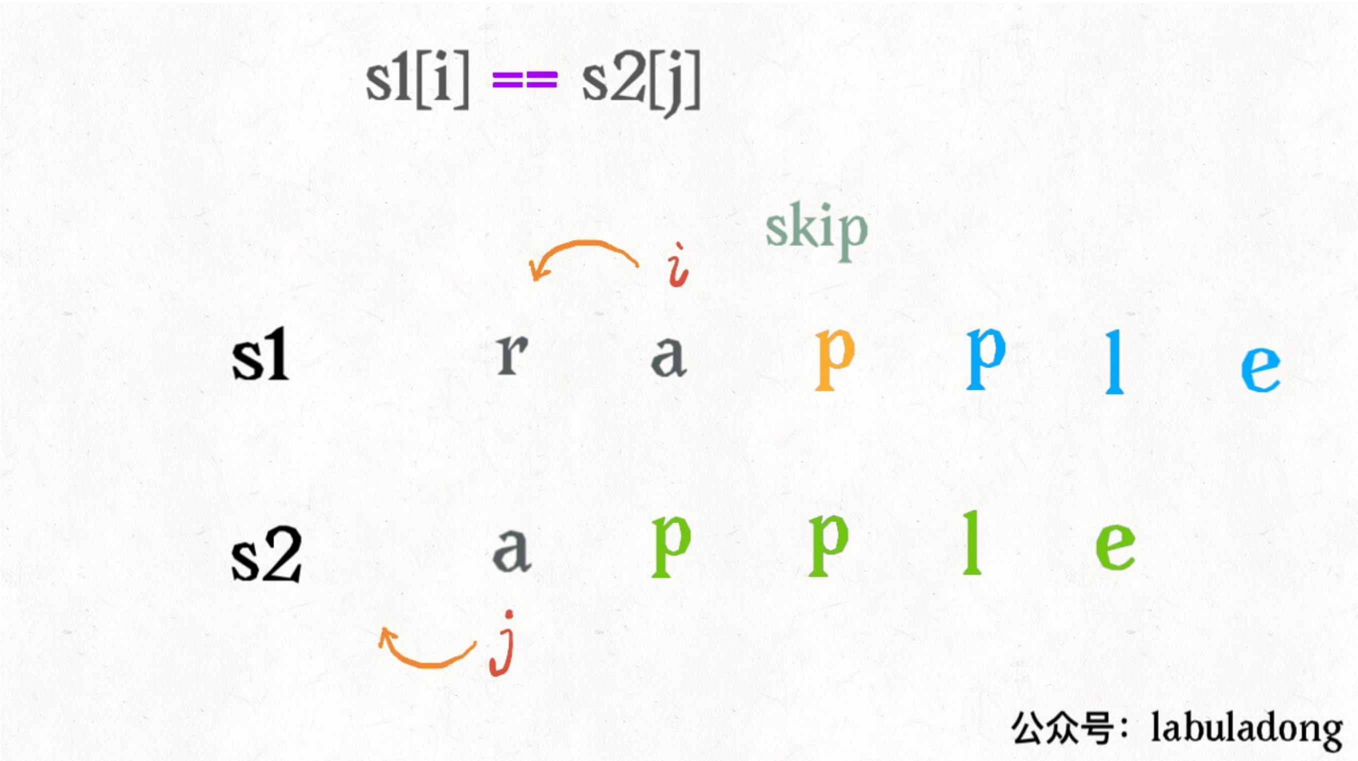
设两个字符串分别为 `"rad"` 和 `"apple"`，为了把 `s1` 变成 `s2`，算法会这样进行：





请记住这个 GIF 过程，这样就能算出编辑距离。关键在于如何做出正确的操作，稍后会讲。

根据上面的 GIF，可以发现操作不只有三个，其实还有第四个操作，就是什么都不要做（skip）。比如这个情况：



因为这两个字符本来就相同，为了使编辑距离最小，显然不应该对它们有任何操作，直接往前移动 `i,j` 即可。

还有一个很容易处理的情况，就是 `j` 走完 `s2` 时，如果 `i` 还没走完 `s1`，那么只能用删除操作把 `s1` 缩短为 `s2`。比如这个情况：



类似的，如果 i 走完 s1 时 j 还没走完了 s2，那就只能用插入操作把 s2 剩下的字符全部插入 s1。等会会看到，这两种情况就是算法的 **base case**。

下面详解一下如何将思路转换成代码，坐稳，要发车了。

二、代码详解

先梳理一下之前的思路：

base case 是 i 走完 s1 或 j 走完 s2，可以直接返回另一个字符串剩下的长度。

对于每对儿字符 s1[i] 和 s2[j]，可以有四种操作：

```
if s1[i] == s2[j]:
    啥都别做（skip）
    i, j 同时向前移动
else:
    三选一：
        插入（insert）
        删除（delete）
        替换（replace）
```

有这个框架，问题就已经解决了。读者也许会问，这个「三选一」到底该怎么选择呢？很简单，全试一遍，哪个操作最后得到的编辑距离最小，就选谁。这里需要递归技巧，理解需要点技巧，先看下代码：

```
def minDistance(s1, s2) -> int:
    # 定义：dp(i, j) 返回 s1[0..i] 和 s2[0..j] 的最小编辑距离
    def dp(i, j):
        # base case
```

```
        if i == -1: return j + 1
        if j == -1: return i + 1

        if s1[i] == s2[j]:
            return dp(i - 1, j - 1)  # 啥都不做
        else:
            return min(
                dp(i, j - 1) + 1,    # 插入
                dp(i - 1, j) + 1,    # 删除
                dp(i - 1, j - 1) + 1 # 替换
            )

    # i, j 初始化指向最后一个索引
    return dp(len(s1) - 1, len(s2) - 1)
```

下面来详细解释一下这段递归代码，base case 应该不用解释了，主要解释一下递归部分。

都说递归代码的可解释性很好，这是有道理的，只要理解函数的定义，就能很清楚地理解算法的逻辑。我们这里 `dp` 函数的定义是这样的：

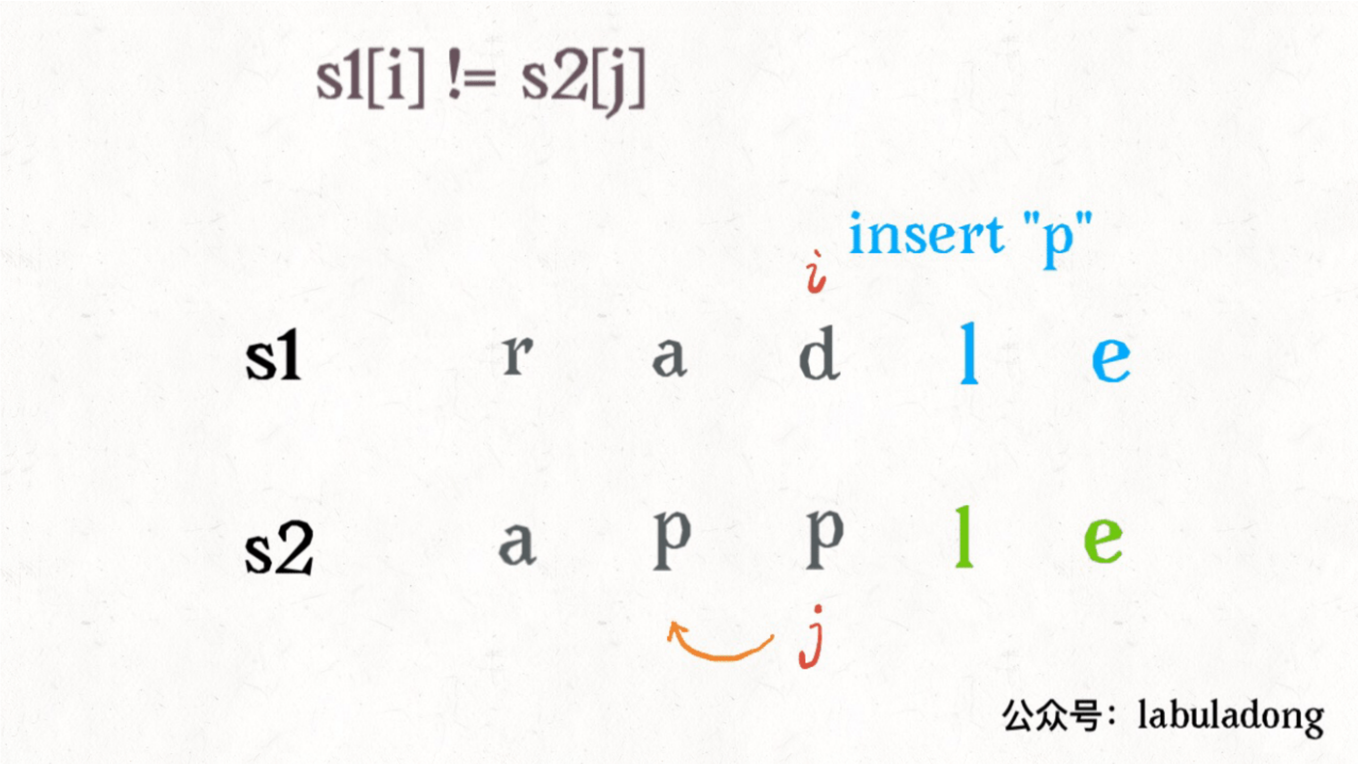
```
def dp(i, j) -> int
# 返回 s1[0..i] 和 s2[0..j] 的最小编辑距离
```

记住这个定义之后，先来看这段代码：

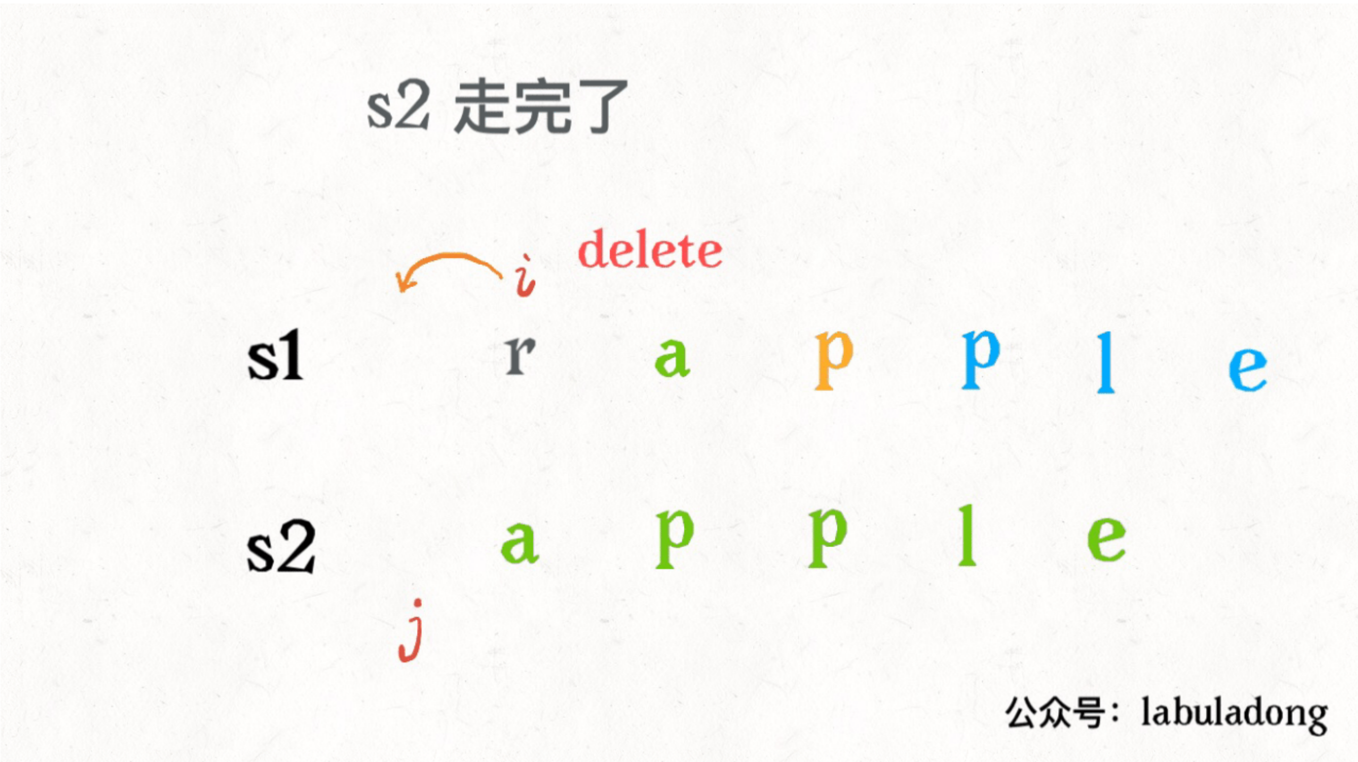
```
if s1[i] == s2[j]:
    return dp(i - 1, j - 1)  # 啥都不做
# 解释：
# 本来就相等，不需要任何操作
# s1[0..i] 和 s2[0..j] 的最小编辑距离等于
# s1[0..i-1] 和 s2[0..j-1] 的最小编辑距离
# 也就是说 dp(i, j) 等于 dp(i-1, j-1)
```

如果 `s1[i]! =s2[j]`，就要对三个操作递归了，稍微需要点思考：

```
dp(i, j - 1) + 1,    # 插入
# 解释：
# 我直接在 s1[i] 插入一个和 s2[j] 一样的字符
# 那么 s2[j] 就被匹配了，前移 j，继续跟 i 对比
# 别忘了操作数加一
```

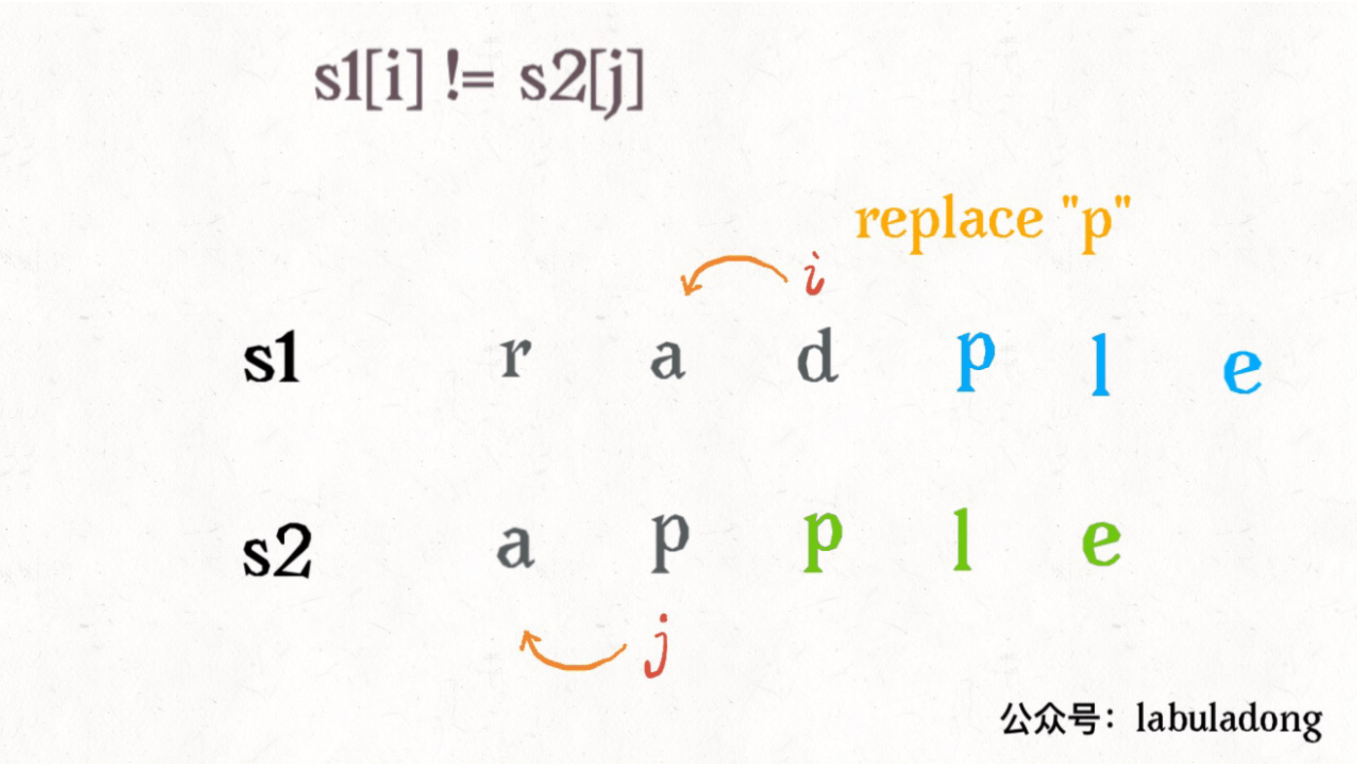



```
dp(i - 1, j) + 1,    # 删除
# 解释：
# 我直接把 s[i] 这个字符删掉
# 前移 i，继续跟 j 对比
# 操作数加一
```




```
dp(i - 1, j - 1) + 1 # 替换
# 解释：
```

```
    # 我直接把 s1[i] 替换成 s2[j]，这样它俩就匹配了
    # 同时前移 i，j 继续对比
    # 操作数加一
```



现在，你应该完全理解这段短小精悍的代码了。还有点小问题就是，这个解法是暴力解法，存在重叠子问题，需要用动态规划技巧来优化。

怎么能一眼看出存在重叠子问题呢？前文 动态规划之正则表达式 有提过，这里再简单提一下，需要抽象出本文算法的递归框架：

```
def dp(i, j):
    dp(i - 1, j - 1) #1
    dp(i, j - 1)     #2
    dp(i - 1, j)     #3
```

对于子问题 `dp(i-1, j-1)`，如何通过原问题 `dp(i, j)` 得到呢？有不止一条路径，比如 `dp(i, j) -> #1` 和 `dp(i, j) -> #2 -> #3`。一旦发现一条重复路径，就说明存在巨量重复路径，也就是重叠子问题。

三、动态规划优化

对于重叠子问题呢，前文 动态规划详解 详细介绍过，优化方法无非是备忘录或者 DP table。

备忘录很好加，原来的代码稍加修改即可：

```
def minDistance(s1, s2) -> int:
    # 备忘录
    memo = dict()
    def dp(i, j):
```

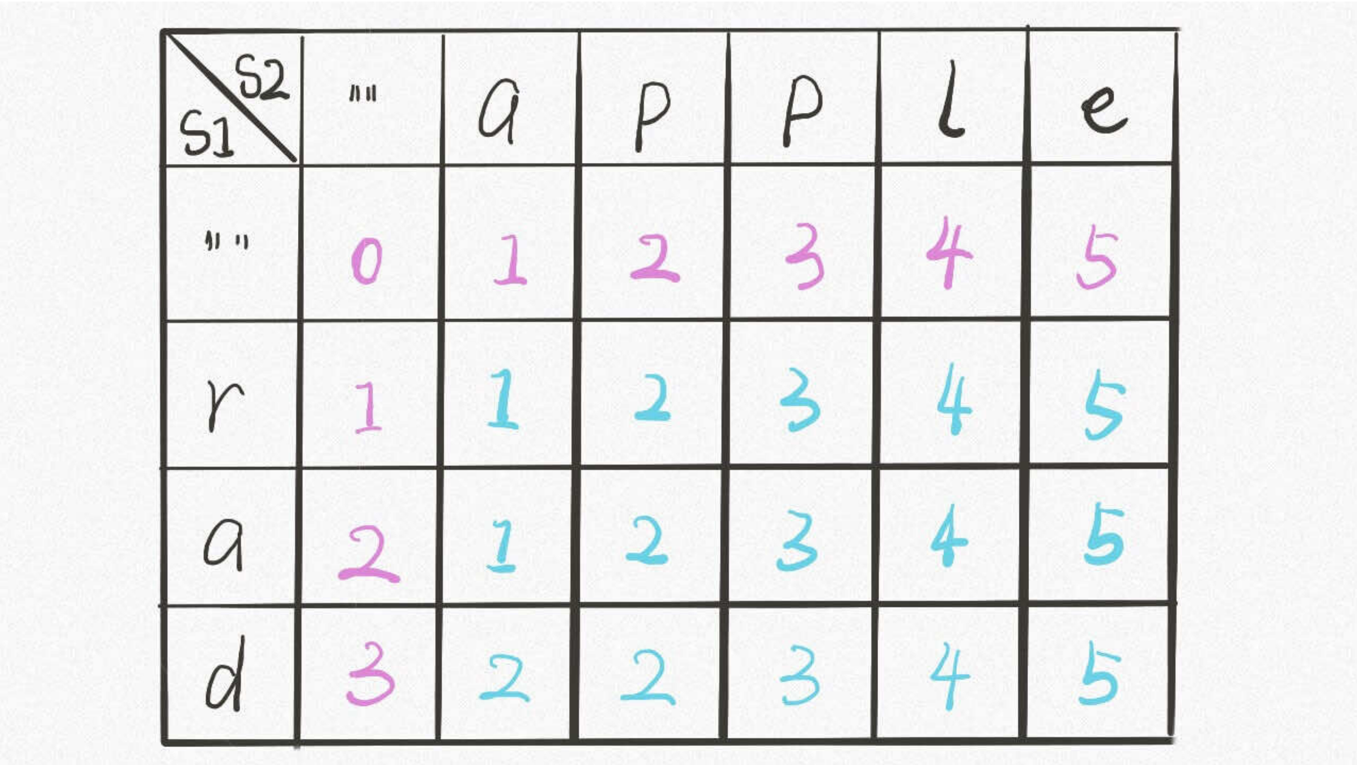
```
        if (i, j) in memo:
            return memo[(i, j)]
        ...

        if s1[i] == s2[j]:
            memo[(i, j)] = ...
        else:
            memo[(i, j)] = ...
        return memo[(i, j)]

    return dp(len(s1) - 1, len(s2) - 1)
```

主要说下 DP table 的解法：

首先明确 dp 数组的含义，dp 数组是一个二维数组，长这样：



有了之前递归解法的铺垫，应该很容易理解。`dp[..][0]` 和 `dp[0][..]` 对应 base case，`dp[i][j]` 的含义和之前的 dp 函数类似：

```
def dp(i, j) -> int
# 返回 s1[0..i] 和 s2[0..j] 的最小编辑距离

dp[i-1][j-1]
# 存储 s1[0..i] 和 s2[0..j] 的最小编辑距离
```

dp 函数的 base case 是 `i, j` 等于 -1，而数组索引至少是 0，所以 dp 数组会偏移一位。

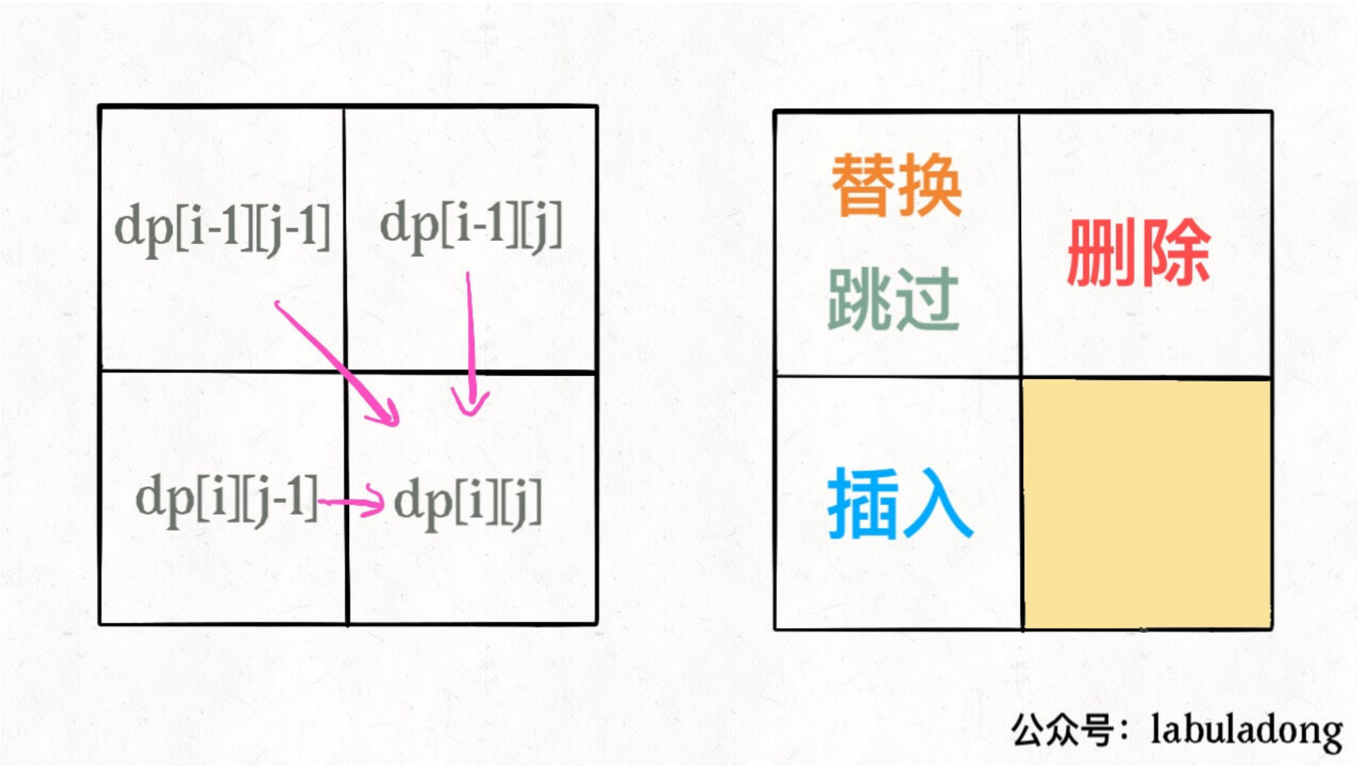
既然 dp 数组和递归 dp 函数含义一样，也就可以直接套用之前的思路写代码，**唯一不同的是，DP table 是自底向上求解，递归解法是自顶向下求解**：

```java
int minDistance(String s1, String s2) {
    int m = s1.length(), n = s2.length();
    // 定义：s1[0..i] 和 s2[0..j] 的最小编辑距离是 dp[i-1][j-1]
    int[][] dp = new int[m + 1][n + 1];
    // base case
    for (int i = 1; i <= m; i++)
        dp[i][0] = i;
    for (int j = 1; j <= n; j++)
        dp[0][j] = j;
    // 自底向上求解
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1.charAt(i-1) == s2.charAt(j-1)) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = min(
                    dp[i - 1][j] + 1,
                    dp[i][j - 1] + 1,
                    dp[i - 1][j - 1] + 1
                );
            }
        }
    }
    // 储存着整个 s1 和 s2 的最小编辑距离
    return dp[m][n];
}

int min(int a, int b, int c) {
    return Math.min(a, Math.min(b, c));
}
```

三、扩展延伸

一般来说，处理两个字符串的动态规划问题，都是按本文的思路处理，建立 DP table。为什么呢，因为易于找出状态转移的关系，比如编辑距离的 DP table：



还有一个细节，既然每个 `dp[i][j]` 只和它附近的三个状态有关，空间复杂度是可以压缩成 `O(min(M, N))` 的（M，N 是两个字符串的长度）。不难，但是可解释性大大降低，读者可以自己尝试优化一下。

你可能还会问，**这里只求出了最小的编辑距离，那具体的操作是什么**？你之前举的修改公众号文章的例子，只有一个最小编辑距离肯定不够，还得知道具体怎么修改才行。

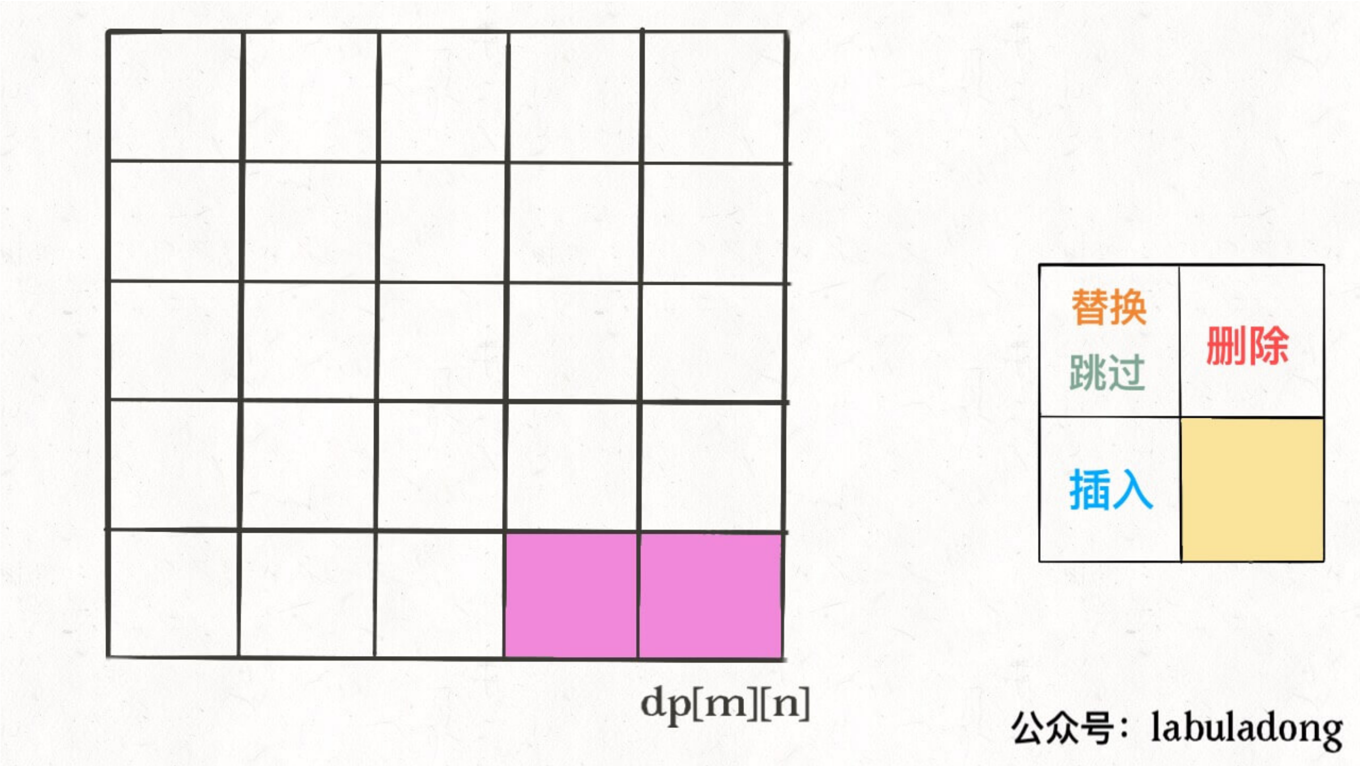
这个其实很简单，代码稍加修改，给 dp 数组增加额外的信息即可：

```java
// int[][] dp;
Node[][] dp;

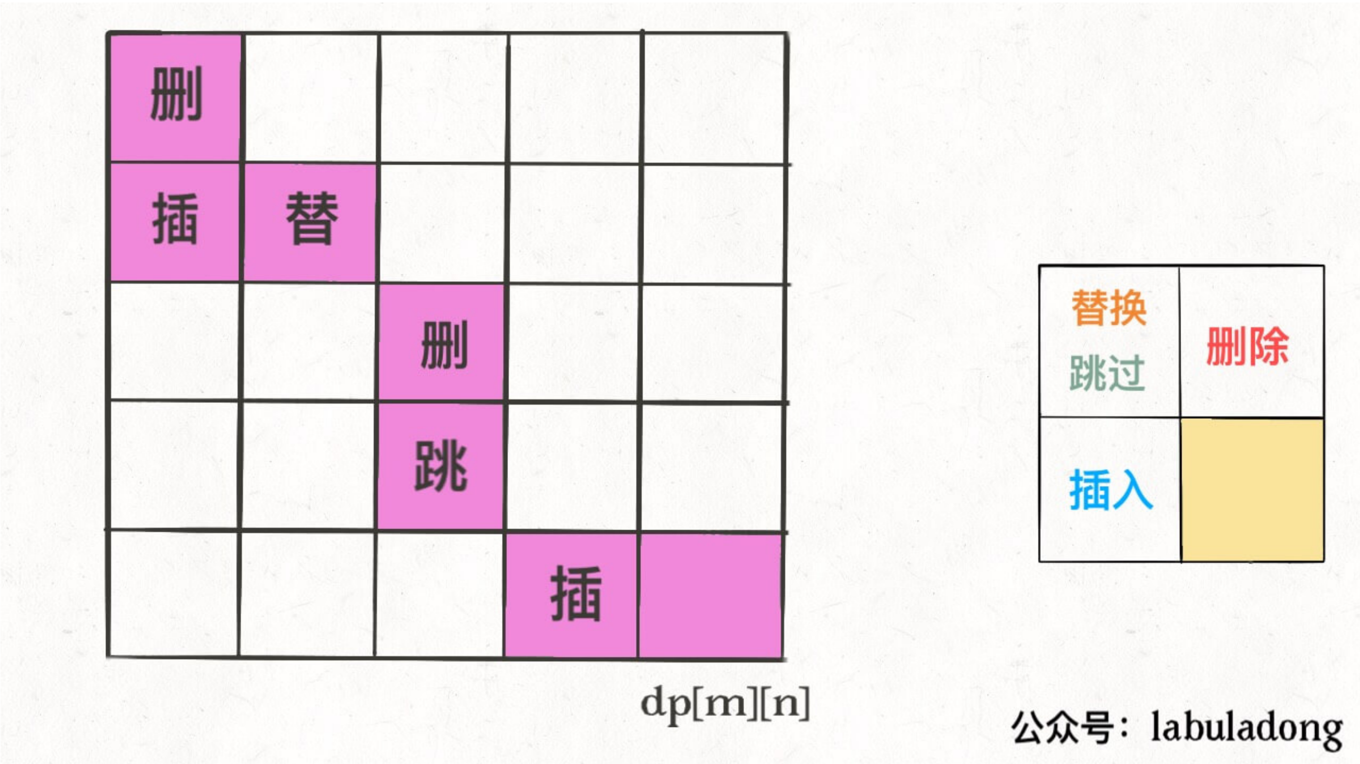
class Node {
    int val;
    int choice;
    // 0 代表啥都不做
    // 1 代表插入
    // 2 代表删除
    // 3 代表替换
}
```

`val` 属性就是之前的 dp 数组的数值，`choice` 属性代表操作。在做最优选择时，顺便把操作记录下来，然后就从结果反推具体操作。

我们的最终结果不是 `dp[m][n]` 吗，这里的 `val` 存着最小编辑距离，`choice` 存着最后一个操作，比如说是插入操作，那么就可以左移一格：



重复此过程，可以一步步回到起点 `dp[0][0]`，形成一条路径，按这条路径上的操作进行编辑，就是最佳方案。



如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

——————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

正则表达式问题

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

10. 正则表达式匹配（困难）

正则表达式是一个非常强力的工具，本文就来具体看一看正则表达式的底层原理是什么。力扣第 10 题「正则表达式匹配」就要求我们实现一个简单的正则匹配算法，包括「.」通配符和「*」通配符。

这两个通配符是最常用的，其中点号「.」可以匹配任意一个字符，星号「*」可以让之前的那个字符重复任意次数（包括 0 次）。

比如说模式串 `".a*b"` 就可以匹配文本 `"zaaab"`，也可以匹配 `"cb"`；模式串 `"a..b"` 可以匹配文本 `"amnb"`；而模式串 `".*"` 就比较牛逼了，它可以匹配任何文本。

题目会给我们输入两个字符串 `s` 和 `p`，`s` 代表文本，`p` 代表模式串，请你判断模式串 `p` 是否可以匹配文本 `s`。我们可以假设模式串只包含小写字母和上述两种通配符且一定合法，不会出现 `*a` 或者 `b**` 这种不合法的模式串，

函数签名如下：

```
bool isMatch(string s, string p);
```

对于我们将要实现的这个正则表达式，难点在那里呢？

点号通配符其实很好实现，`s` 中的任何字符，只要遇到 `.` 通配符，无脑匹配就完事了。主要是这个星号通配符不好实现，一旦遇到 `*` 通配符，前面的那个字符可以选择重复一次，可以重复多次，也可以一次都不出现，这该怎么办？

对于这个问题，答案很简单，对于所有可能出现的情况，全部穷举一遍，只要有一种情况可以完成匹配，就认为 `p` 可以匹配 `s`。那么一旦涉及两个字符串的穷举，我们就应该条件反射地想到动态规划的技巧了。

一、思路分析

我们先脑补一下，`s` 和 `p` 相互匹配的过程大致是，两个指针 `i, j` 分别在 `s` 和 `p` 上移动，如果最后两个指针都能移动到字符串的末尾，那么久匹配成功，反之则匹配失败。

正则表达算法问题只需要把住一个基本点：看两个字符是否匹配，一切逻辑围绕匹配/不匹配两种情况展开即可。

如果不考虑 `*` 通配符，面对两个待匹配字符 `s[i]` 和 `p[j]`，我们唯一能做的就是看他俩是否匹配：

```
bool isMatch(string s, string p) {
    int i = 0, j = 0;
    while (i < s.size() && j < p.size()) {
        // 「.」通配符就是万金油
        if (s[i] == p[j] || p[j] == '.') {
            // 匹配，接着匹配 s[i+1..] 和 p[j+1..]
            i++; j++;
        } else {
            // 不匹配
            return false;
        }
    }
    return i == j;
}
```

那么考虑一下，如果加入 `*` 通配符，局面就会稍微复杂一些，不过只要分情况来分析，也不难理解。

当 `p[j + 1]` 为 `*` 通配符时，我们分情况讨论下：

1、如果 `s[i] == p[j]`，那么有两种情况：

1.1 `p[j]` 有可能会匹配多个字符，比如 `s = "aaa", p = "a*"`，那么 `p[0]` 会通过 `*` 匹配 3 个字符 `"a"`。

1.2 `p[i]` 也有可能匹配 0 个字符，比如 `s = "aa", p = "a*aa"`，由于后面的字符可以匹配 `s`，所以 `p[0]` 只能匹配 0 次。

2、如果 `s[i] != p[j]`，只有一种情况：

`p[j]` 只能匹配 0 次，然后看下一个字符是否能和 `s[i]` 匹配。比如说 `s = "aa", p = "b*aa"`，此时 `p[0]` 只能匹配 0 次。

综上，可以把之前的代码针对 `*` 通配符进行一下改造：

```
if (s[i] == p[j] || p[j] == '.') {
    // 匹配
    if (j < p.size() - 1 && p[j + 1] == '*') {
        // 有 * 通配符，可以匹配 0 次或多次
    } else {
        // 无 * 通配符，老老实实匹配 1 次
        i++; j++;
    }
} else {
    // 不匹配
    if (j < p.size() - 1 && p[j + 1] == '*') {
        // 有 * 通配符，只能匹配 0 次
    } else {
```

```
            // 无 * 通配符，匹配无法进行下去了
            return false;
        }
    }
```

整体的思路已经很清晰了，但现在的问题是，遇到 `*` 通配符时，到底应该匹配 0 次还是匹配多次？多次是几次？

你看，这就是一个做「选择」的问题，要把所有可能的选择都穷举一遍才能得出结果。动态规划算法的核心就是「状态」和「选择」，**「状态」无非就是 `i` 和 `j` 两个指针的位置，「选择」就是 `p[j]` 选择匹配几个字符。**

二、动态规划解法

根据「状态」，我们可以定义一个 `dp` 函数：

```
bool dp(string& s, int i, string& p, int j);
```

————————————————

应合作方要求，本文不便在此发布，请扫码关注回复关键词「正则」查看：

4.3 背包问题

背包问题是一类经典的动态规划问题。

当我说 XXX 问题是一类经典动态规划问题的时候，并不是指题目的形式经典，而是强调该类问题的状态转移方程非常有特点，其「状态」和「选择」的定义遵循一定的模式，可以抽象为一类特定的问题。

你之后会看到大部分题目都会把题目原型深深地隐藏起来，缺乏思考的话甚至看不出来这题竟然是一道背包问题，但一旦你发现一道题目可以被抽象为背包问题，那么你就可以按照背包问题的解题思路写出标准化的答案。

公众号标签：背包问题

0-1 背包问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

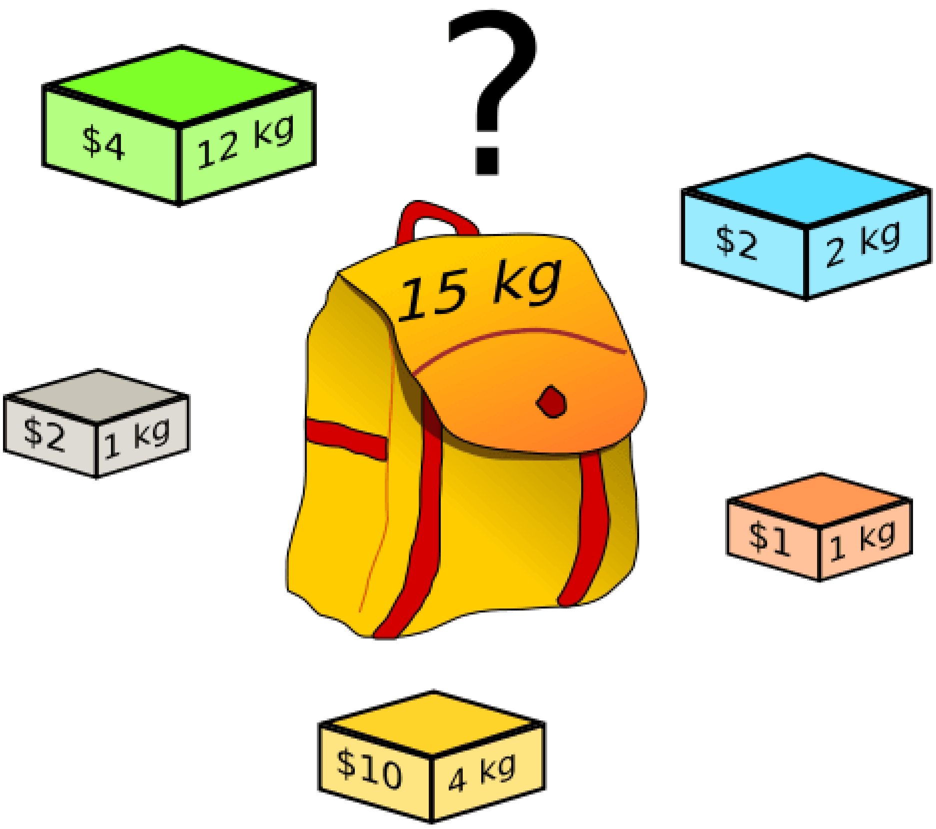
学算法认准 labuladong，致力于把算法讲清楚！网页版 [点这里](#)。

本文有视频版：[0-1背包问题详解](#)

后台天天有人问背包问题，这个问题其实不难啊，如果我们号动态规划系列的十几篇文章你都看过，借助框架，遇到背包问题可以说是手到擒来好吧。无非就是状态 + 选择，也没啥特别之处嘛。

今天就来说一下背包问题吧，就讨论最常说的 0-1 背包问题。描述：

给你一个可装载重量为 `W` 的背包和 `N` 个物品，每个物品有重量和价值两个属性。其中第 `i` 个物品的重量为 `wt[i]`，价值为 `val[i]`，现在让你用这个背包装物品，最多能装的价值是多少？



举个简单的例子，输入如下：

```
N = 3, W = 4
wt = [2, 1, 3]
```

```
val = [4, 2, 3]
```

算法返回 6，选择前两件物品装进背包，总重量 3 小于 `W`，可以获得最大价值 6。

题目就是这么简单，一个典型的动态规划问题。这个题目中的物品不可以分割，要么装进包里，要么不装，不能说切成两块装一半。这就是 0-1 背包这个名词的来历。

解决这个问题没有什么排序之类巧妙的方法，只能穷举所有可能，根据我们 动态规划详解 中的套路，直接走流程就行了。

动规标准套路

看来我得每篇动态规划文章都得重复一遍套路，历史文章中的动态规划问题都是按照下面的套路来的。

第一步要明确两点，「状态」和「选择」。

先说状态，如何才能描述一个问题局面？只要给几个物品和一个背包的容量限制，就形成了一个背包问题呀。**所以状态有两个，就是「背包的容量」和「可选择的物品」**。

再说选择，也很容易想到啊，对于每件物品，你能选择什么？**选择就是「装进背包」或者「不装进背包」嘛**。

明白了状态和选择，动态规划问题基本上就解决了，只要往这个框架套就完事儿了：

```
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 择优(选择1，选择2...)
```

> PS：此框架出自历史文章 团灭 LeetCode 股票问题。

第二步要明确 `dp` 数组的定义。

首先看看刚才找到的「状态」，有两个，也就是说我们需要一个二维 `dp` 数组。

`dp[i][w]` 的定义如下：对于前 `i` 个物品，当前背包的容量为 `w`，这种情况下可以装的最大价值是 `dp[i][w]`。

比如说，如果 `dp[3][5] = 6`，其含义为：对于给定的一系列物品中，若只对前 3 个物品进行选择，当背包容量为 5 时，最多可以装下的价值为 6。

> PS：为什么要这么定义？便于状态转移，或者说这就是套路，记下来就行了。建议看一下我们的动态规划系列文章，几种套路都被扒得清清楚楚了。

根据这个定义，我们想求的最终答案就是 `dp[N][W]`。base case 就是 `dp[0][..] = dp[..][0] = 0`，因为没有物品或者背包没有空间的时候，能装的最大价值就是 0。

细化上面的框架：

```
int[][] dp[N+1][W+1]
dp[0][..] = 0
dp[..][0] = 0

for i in [1..N]:
    for w in [1..W]:
        dp[i][w] = max(
            把物品 i 装进背包,
            不把物品 i 装进背包
        )
return dp[N][W]
```

第三步，根据「选择」，思考状态转移的逻辑。

简单说就是，上面伪码中「把物品 `i` 装进背包」和「不把物品 `i` 装进背包」怎么用代码体现出来呢？

这就要结合对 `dp` 数组的定义，看看这两种选择会对状态产生什么影响：

先重申一下刚才我们的 `dp` 数组的定义：

`dp[i][w]` 表示：对于前 `i` 个物品，当前背包的容量为 `w` 时，这种情况下可以装下的最大价值是 `dp[i][w]`。

如果你没有把这第 `i` 个物品装入背包，那么很显然，最大价值 `dp[i][w]` 应该等于 `dp[i-1][w]`，继承之前的结果。

如果你把这第 `i` 个物品装入了背包，那么 `dp[i][w]` 应该等于 `dp[i-1][w - wt[i-1]] + val[i-1]`。

首先，由于 `i` 是从 1 开始的，所以 `val` 和 `wt` 的索引是 `i-1` 时表示第 `i` 个物品的价值和重量。

而 `dp[i-1][w - wt[i-1]]` 也很好理解：你如果装了第 `i` 个物品，就要寻求剩余重量 `w - wt[i-1]` 限制下的最大价值，加上第 `i` 个物品的价值 `val[i-1]`。

综上就是两种选择，我们都已经分析完毕，也就是写出来了状态转移方程，可以进一步细化代码：

```
for i in [1..N]:
    for w in [1..W]:
        dp[i][w] = max(
            dp[i-1][w],
            dp[i-1][w - wt[i-1]] + val[i-1]
        )
return dp[N][W]
```

最后一步，把伪码翻译成代码，处理一些边界情况。

我用 Java 写的代码，把上面的思路完全翻译了一遍，并且处理了 `w - wt[i-1]` 可能小于 0 导致数组索引越界的问题：

```java
int knapsack(int W, int N, int[] wt, int[] val) {
    // base case 已初始化
    int[][] dp = new int[N + 1][W + 1];
    for (int i = 1; i <= N; i++) {
        for (int w = 1; w <= W; w++) {
            if (w - wt[i - 1] < 0) {
                // 这种情况下只能选择不装入背包
                dp[i][w] = dp[i - 1][w];
            } else {
                // 装入或者不装入背包，择优
                dp[i][w] = Math.max(
                    dp[i - 1][w - wt[i-1]] + val[i-1],
                    dp[i - 1][w]
                );
            }
        }
    }

    return dp[N][W];
}
```

至此，背包问题就解决了，相比而言，我觉得这是比较简单的动态规划问题，因为状态转移的推导比较自然，基本上你明确了 `dp` 数组的定义，就可以理所当然地确定状态转移了。

接下来可阅读：

- 完全背包问题之零钱兑换
- 背包问题变体之子集分割

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

完全背包问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

518. 零钱兑换II（中等）

零钱兑换 2 是另一种典型背包问题的变体，我们前文已经讲了 经典动态规划：0-1 背包问题 和 背包问题变体：相等子集分割。

读本文之前，希望你已经看过前两篇文章，看过了动态规划和背包问题的套路，这篇继续按照背包问题的套路，列举一个背包问题的变形。

本文聊的是力扣第 518 题「零钱兑换 II」，我描述一下题目：

给定不同面额的硬币 `coins` 和一个总金额 `amount`，写一个函数来计算可以凑成总金额的硬币组合数。**假设每一种面额的硬币有无限个。**

我们要完成的函数的签名如下：

```
int change(int amount, int[] coins);
```

比如说输入 `amount = 5, coins = [1,2,5]`，算法应该返回 4，因为有如下 4 种方式可以凑出目标金额：

5=5

5=2+2+1

5=2+1+1+1

5=1+1+1+1+1

如果输入的 `amount = 5, coins = [3]`，算法应该返回 0，因为用面额为 3 的硬币无法凑出总金额 5。

> PS：至于零钱兑换 I，在我们前文 动态规划套路详解 写过。

我们可以把这个问题转化为背包问题的描述形式：

有一个背包，最大容量为 amount，有一系列物品 coins，每个物品的重量为 coins[i]，**每个物品的数量无限**。请问有多少种方法，能够把背包恰好装满？

这个问题和我们前面讲过的两个背包问题，有一个最大的区别就是，每个物品的数量是无限的，这也就是传说中的「**完全背包问题**」，没啥高大上的，无非就是状态转移方程有一点变化而已。

下面就以背包问题的描述形式，继续按照流程来分析。

解题思路

第一步要明确两点，「状态」和「选择」。

状态有两个，就是「背包的容量」和「可选择的物品」，选择就是「装进背包」或者「不装进背包」嘛，背包问题的套路都是这样。

明白了状态和选择，动态规划问题基本上就解决了，只要往这个框架套就完事儿了：

```
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 计算(选择1，选择2...)
```

第二步要明确 dp 数组的定义。

首先看看刚才找到的「状态」，有两个，也就是说我们需要一个二维 dp 数组。

dp[i][j] 的定义如下：

若只使用前 i 个物品（可以重复使用），当背包容量为 j 时，有 dp[i][j] 种方法可以装满背包。

换句话说，翻译回我们题目的意思就是：

若只使用 coins 中的前 i 个硬币的面值，若想凑出金额 j，有 dp[i][j] 种凑法。

经过以上的定义，可以得到：

base case 为 dp[0][..] = 0， dp[..][0] = 1。因为如果不使用任何硬币面值，就无法凑出任何金额；如果凑出的目标金额为 0，那么"无为而治"就是唯一的一种凑法。

我们最终想得到的答案就是 dp[N][amount]，其中 N 为 coins 数组的大小。

大致的伪码思路如下：

```
int dp[N+1][amount+1]
dp[0][..] = 0
dp[..][0] = 1

for i in [1..N]:
    for j in [1..amount]:
        把物品 i 装进背包,
```

```
         不把物品 i 装进背包
return dp[N][amount]
```

第三步，根据「选择」，思考状态转移的逻辑。

注意，我们这个问题的特殊点在于物品的数量是无限的，所以这里和之前写的 0-1 背包问题 文章有所不同。

如果你不把这第 `i` 个物品装入背包，也就是说你不使用 `coins[i]` 这个面值的硬币，那么凑出面额 `j` 的方法数 `dp[i][j]` 应该等于 `dp[i-1][j]`，继承之前的结果。

如果你把这第 `i` 个物品装入了背包，也就是说你使用 `coins[i]` 这个面值的硬币，那么 `dp[i][j]` 应该等于 `dp[i][j-coins[i-1]]`。

首先由于 `i` 是从 1 开始的，所以 `coins` 的索引是 `i-1` 时表示第 `i` 个硬币的面值。

`dp[i][j-coins[i-1]]` 也不难理解，如果你决定使用这个面值的硬币，那么就应该关注如何凑出金额 `j - coins[i-1]`。

比如说，你想用面值为 2 的硬币凑出金额 5，那么如果你知道了凑出金额 3 的方法，再加上一枚面额为 2 的硬币，不就可以凑出 5 了嘛。

综上就是两种选择，而我们想求的 `dp[i][j]` 是「共有多少种凑法」，所以 `dp[i][j]` 的值应该是以上两种选择的结果之和：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= amount; j++) {
        if (j - coins[i-1] >= 0)
            dp[i][j] = dp[i - 1][j]
                     + dp[i][j-coins[i-1]];
return dp[N][W]
```

> PS：有的读者在这里可能会有疑问，不是说可以重复使用硬币吗？那么如果我确定「使用第 `i` 个面值的硬币」，我怎么确定这个面值的硬币被使用了多少枚？简单的一个 `dp[i][j-coins[i-1]]` 可以包含重复使用第 `i` 个硬币的情况吗？

对于这个问题，建议你再仔回头细阅读一下我们对 `dp` 数组的定义，然后把这个定义代入 `dp[i][j-coins[i-1]]` 看看：

若只使用前 `i` 个物品（可以重复使用），当背包容量为 `j-coins[i-1]` 时，有 `dp[i][j-coins[i-1]]` 种方法可以装满背包。

看到了吗，`dp[i][j-coins[i-1]]` 也是允许你使用第 `i` 个硬币的，所以说已经包含了重复使用硬币的情况，你一百个放心。

最后一步，把伪码翻译成代码，处理一些边界情况。

我用 Java 写的代码，把上面的思路完全翻译了一遍，并且处理了一些边界问题：

```java
int change(int amount, int[] coins) {
    int n = coins.length;
    int[][] dp = int[n + 1][amount + 1];
    // base case
    for (int i = 0; i <= n; i++)
        dp[i][0] = 1;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= amount; j++)
            if (j - coins[i-1] >= 0)
                dp[i][j] = dp[i - 1][j]
                         + dp[i][j - coins[i-1]];
            else
                dp[i][j] = dp[i - 1][j];
    }
    return dp[n][amount];
}
```

而且，我们通过观察可以发现，`dp` 数组的转移只和 `dp[i][..]` 和 `dp[i-1][..]` 有关，所以可以压缩状态，进一步降低算法的空间复杂度：

```java
int change(int amount, int[] coins) {
    int n = coins.length;
    int[] dp = new int[amount + 1];
    dp[0] = 1; // base case
    for (int i = 0; i < n; i++)
        for (int j = 1; j <= amount; j++)
            if (j - coins[i] >= 0)
                dp[j] = dp[j] + dp[j-coins[i]];

    return dp[amount];
}
```

这个解法和之前的思路完全相同，将二维 `dp` 数组压缩为一维，时间复杂度 O(N*amount)，空间复杂度 O(amount)。

至此，这道零钱兑换问题也通过背包问题的框架解决了。

接下来可阅读：

- 背包问题变体之子集分割

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

子集背包问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

416. 分割等和子集（中等）

上篇文章 经典动态规划：0-1 背包问题 详解了通用的 0-1 背包问题，今天来看看背包问题的思想能够如何运用到其他算法题目。

而且，不是经常有读者问，怎么将二维动态规划压缩成一维动态规划吗？这就是空间压缩，很容易的，本文也会提及这种技巧。

读者在阅读本文之前务必读懂前文 经典动态规划：0-1 背包问题 中讲的套路，因为本文就是按照背包问题的解题模板来讲解的。

一、问题分析

看一下力扣第 416 题「分割等和子集」：

输入一个只包含正整数的非空数组 `nums`，请你写一个算法，判断这个数组是否可以被分割成两个子集，使得两个子集的元素和相等。

算法的函数签名如下：

```
// 输入一个集合，返回是否能够分割成和相等的两个子集
boolean canPartition(int[] nums);
```

比如说输入 `nums = [1,5,11,5]`，算法返回 true，因为 `nums` 可以分割成 `[1,5,5]` 和 `[11]` 这两个子集。

如果说输入 `nums = [1,3,2,5]`，算法返回 false，因为 `nums` 无论如何都不能分割成两个和相等的子集。

对于这个问题，看起来和背包没有任何关系，为什么说它是背包问题呢？

首先回忆一下背包问题大致的描述是什么：

给你一个可装载重量为 `W` 的背包和 `N` 个物品，每个物品有重量和价值两个属性。其中第 `i` 个物品的重量为 `wt[i]`，价值为 `val[i]`，现在让你用这个背包装物品，最多能装的价值是多少？

那么对于这个问题，我们可以先对集合求和，得出 `sum`，把问题转化为背包问题：

给一个可装载重量为 `sum / 2` 的背包和 `N` 个物品，每个物品的重量为 `nums[i]`。现在让你装物品，是否存在一种装法，能够恰好将背包装满？

你看，这就是背包问题的模型，甚至比我们之前的经典背包问题还要简单一些，**下面我们就直接转换成背包问题**，开始套前文讲过的背包问题框架即可。

二、解法分析

第一步要明确两点，「状态」和「选择」。

这个前文 经典动态规划：背包问题 已经详细解释过了，状态就是「背包的容量」和「可选择的物品」，选择就是「装进背包」或者「不装进背包」。

第二步要明确 `dp` 数组的定义。

按照背包问题的套路，可以给出如下定义：

`dp[i][j] = x` 表示，对于前 `i` 个物品，当前背包的容量为 `j` 时，若 `x` 为 `true`，则说明可以恰好将背包装满，若 `x` 为 `false`，则说明不能恰好将背包装满。

比如说，如果 `dp[4][9] = true`，其含义为：对于容量为 9 的背包，若只是用前 4 个物品，可以有一种方法把背包恰好装满。

或者说对于本题，含义是对于给定的集合中，若只对前 4 个数字进行选择，存在一个子集的和可以恰好凑出 9。

根据这个定义，我们想求的最终答案就是 `dp[N][sum/2]`，base case 就是 `dp[..][0] = true` 和 `dp[0][..] = false`，因为背包没有空间的时候，就相当于装满了，而当没有物品可选择的时候，肯定没办法装满背包。

第三步，根据「选择」，思考状态转移的逻辑。

回想刚才的 `dp` 数组含义，可以根据「选择」对 `dp[i][j]` 得到以下状态转移：

如果不把 `nums[i]` 算入子集，**或者说你不把这第 `i` 个物品装入背包**，那么是否能够恰好装满背包，取决于上一个状态 `dp[i-1][j]`，继承之前的结果。

如果把 `nums[i]` 算入子集，**或者说你把这第 `i` 个物品装入了背包**，那么是否能够恰好装满背包，取决于状态 `dp[i-1][j-nums[i-1]]`。

首先，由于 `i` 是从 1 开始的，而数组索引是从 0 开始的，所以第 `i` 个物品的重量应该是 `nums[i-1]`，这一点不要搞混。

`dp[i - 1][j-nums[i-1]]` 也很好理解：你如果装了第 `i` 个物品，就要看背包的剩余重量 `j - nums[i-1]` 限制下是否能够被恰好装满。

换句话说，如果 `j - nums[i-1]` 的重量可以被恰好装满，那么只要把第 `i` 个物品装进去，也可恰好装满 `j` 的重量；否则的话，重量 `j` 肯定是装不满的。

最后一步，把伪码翻译成代码，处理一些边界情况。

以下是我的 C++ 代码，完全翻译了之前的思路，并处理了一些边界情况：

```
boolean canPartition(int[] nums) {
    int sum = 0;
    for (int num : nums) sum += num;
    // 和为奇数时，不可能划分成两个和相等的集合
    if (sum % 2 != 0) return false;
    int n = nums.length;
    sum = sum / 2;
    boolean[][] dp = new boolean[n + 1][sum + 1];
    // base case
    for (int i = 0; i <= n; i++)
        dp[i][0] = true;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (j - nums[i - 1] < 0) {
                // 背包容量不足，不能装入第 i 个物品
                dp[i][j] = dp[i - 1][j];
            } else {
                // 装入或不装入背包
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]];
            }
        }
    }
    return dp[n][sum];
}
```

三、进一步优化

再进一步，是否可以优化这个代码呢？**注意到 `dp[i][j]` 都是通过上一行 `dp[i-1][..]` 转移过来的**，之前的数据都不会再使用了。

所以，我们可以 对动态规划进行降维打击，将二维 `dp` 数组压缩为一维，节约空间复杂度：

```
boolean canPartition(int[] nums) {
    int sum = 0;
    for (int num : nums) sum += num;
    // 和为奇数时，不可能划分成两个和相等的集合
    if (sum % 2 != 0) return false;
    int n = nums.length;
    sum = sum / 2;
    boolean[] dp = new boolean[sum + 1];

    // base case
    dp[0] = true;

    for (int i = 0; i < n; i++) {
        for (int j = sum; j >= 0; j--) {
            if (j - nums[i] >= 0) {
```

```
                dp[j] = dp[j] || dp[j - nums[i]];
            }
        }
    }
    return dp[sum];
}
```

其实这段代码和之前的解法思路完全相同，只在一行 `dp` 数组上操作，`i` 每进行一轮迭代，`dp[j]` 其实就相当于 `dp[i-1][j]`，所以只需要一维数组就够用了。

唯一需要注意的是 `j` 应该从后往前反向遍历，因为每个物品（或者说数字）只能用一次，以免之前的结果影响其他的结果。

至此，子集切割的问题就完全解决了，时间复杂度 O(n*sum)，空间复杂度 O(sum)。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

4.4 用动态规划玩游戏



动态规划的底层逻辑也是穷举，只不过动态规划问题具有一些特殊的性质，使得穷举的过程中存在可优化的空间。

这里先提醒你，学习动态规划问题要格外注意这几个词：「**状态**」，「**选择**」，「**dp 数组的定义**」。你把这几个词理解到位了，就理解了动态规划的核心。

当然，动态规划问题的题型非常广泛，我不能保证你理解了核心就能做出所有动态规划题目，但我保证你理解了核心原理之后可以很轻松地理解别人的正确解法。如果自己勤加练习和总结，解决大部分中上难度的动态规划问题应该是没什么问题的。

公众号标签：手把手刷动态规划

团灭 LeetCode 股票买卖问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

121. 买卖股票的最佳时机（简单）

122. 买卖股票的最佳时机 II（简单）

123. 买卖股票的最佳时机 III（困难）

188. 买卖股票的最佳时机 IV（困难）

309. 最佳买卖股票时机含冷冻期（中等）

714. 买卖股票的最佳时机含手续费（中等）

很多读者抱怨 LeetCode 的股票系列问题奇技淫巧太多，如果面试真的遇到这类问题，基本不会想到那些巧妙的办法，怎么办？**所以本文拒绝奇技淫巧，而是稳扎稳打，只用一种通用方法解决所用问题，以不变应万变。**

这篇文章参考 英文版高赞题解 的思路，用状态机的技巧来解决，可以全部提交通过。不要觉得这个名词高大上，文学词汇而已，实际上就是 DP table，看一眼就明白了。

先随便抽出一道题，看看别人的解法：

```
int maxProfit(vector<int>& prices) {
    if(prices.empty()) return 0;
    int s1 = -prices[0], s2 = INT_MIN, s3 = INT_MIN, s4 = INT_MIN;

    for(int i = 1; i < prices.size(); ++i) {
        s1 = max(s1, -prices[i]);
        s2 = max(s2, s1 + prices[i]);
        s3 = max(s3, s2 - prices[i]);
        s4 = max(s4, s3 + prices[i]);
    }
    return max(0, s4);
}
```

能看懂吧？会做了吗？不可能的，你看不懂，这才正常。就算你勉强看懂了，下一个问题你还是做不出来。为什么别人能写出这么诡异却又高效的解法呢？因为这类问题是有框架的，但是人家不会告诉你的，因为一旦告诉你，你五分钟就学会了，该算法题就不再神秘，变得不堪一击了。

本文就来告诉你这个框架，然后带着你一道一道秒杀。这篇文章用状态机的技巧来解决，可以全部提交通过。不要觉得这个名词高大上，文学词汇而已，实际上就是 DP table，看一眼就明白了。

这 6 道题目是有共性的，我们只需要抽出来力扣第 188 题「买卖股票的最佳时机 IV」进行研究，因为这道题是最泛化的形式，其他的问题都是这个形式的简化，看下题目：

188. 买卖股票的最佳时机 IV labuladong 题解 思路

难度 困难 667

给定一个整数数组 `prices` ，它的第 `i` 个元素 `prices[i]` 是一支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 **k** 笔交易。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

```
输入：k = 2, prices = [3,2,6,5,0,3]
输出：7
解释：在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 = 6-2 = 4 。
     随后，在第 5 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出, 这笔交易所能获得利润 = 3-0 = 3 。
```

第一题是只进行一次交易，相当于 `k = 1`；第二题是不限交易次数，相当于 `k = +infinity`（正无穷）；第三题是只进行 2 次交易，相当于 `k = 2`；剩下两道也是不限次数，但是加了交易「冷冻期」和「手续费」的额外条件，其实就是第二题的变种，都很容易处理。

下面言归正传，开始解题。

一、穷举框架

首先，还是一样的思路：如何穷举？

动态规划核心套路 说过，动态规划算法本质上就是穷举「状态」，然后在「选择」中选择最优解。

那么对于这道题，我们具体到每一天，看看总共有几种可能的「状态」，再找出每个「状态」对应的「选择」。我们要穷举所有「状态」，穷举的目的是根据对应的「选择」更新状态。听起来抽象，你只要记住「状态」和「选择」两个词就行，下面实操一下就很容易明白了。

```
for 状态1 in 状态1的所有取值:
    for 状态2 in 状态2的所有取值:
        for ...
            dp[状态1][状态2][...] = 择优(选择1, 选择2...)
```

比如说这个问题，**每天都有三种「选择」**：买入、卖出、无操作，我们用 `buy`, `sell`, `rest` 表示这三种选择。

但问题是，并不是每天都可以任意选择这三种选择的，因为 `sell` 必须在 `buy` 之后，`buy` 必须在 `sell` 之后。那么 `rest` 操作还应该分两种状态，一种是 `buy` 之后的 `rest`（持有了股票），一种是 `sell` 之后的 `rest`（没有持有股票）。而且别忘了，我们还有交易次数 `k` 的限制，就是说你 `buy` 还只能在 `k > 0` 的前提下操作。

很复杂对吧，不要怕，我们现在的目的只是穷举，你有再多的状态，老夫要做的就是一把梭全部列举出来。

这个问题的「状态」有三个，第一个是天数，第二个是允许交易的最大次数，第三个是当前的持有状态（即之前说的 `rest` 的状态，我们不妨用 1 表示持有，0 表示没有持有）。然后我们用一个三维数组就可以装下这几种状态的全部组合：

```
dp[i][k][0 or 1]
0 <= i <= n - 1, 1 <= k <= K
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。
此问题共 n × K × 2 种状态，全部穷举就能搞定。

for 0 <= i < n:
    for 1 <= k <= K:
        for s in {0, 1}:
            dp[i][k][s] = max(buy, sell, rest)
```

而且我们可以用自然语言描述出每一个状态的含义，比如说 `dp[3][2][1]` 的含义就是：今天是第三天，我现在手上持有着股票，至今最多进行 2 次交易。再比如 `dp[2][3][0]` 的含义：今天是第二天，我现在手上没有持有股票，至今最多进行 3 次交易。很容易理解，对吧？

我们想求的最终答案是 `dp[n - 1][K][0]`，即最后一天，最多允许 `K` 次交易，最多获得多少利润。

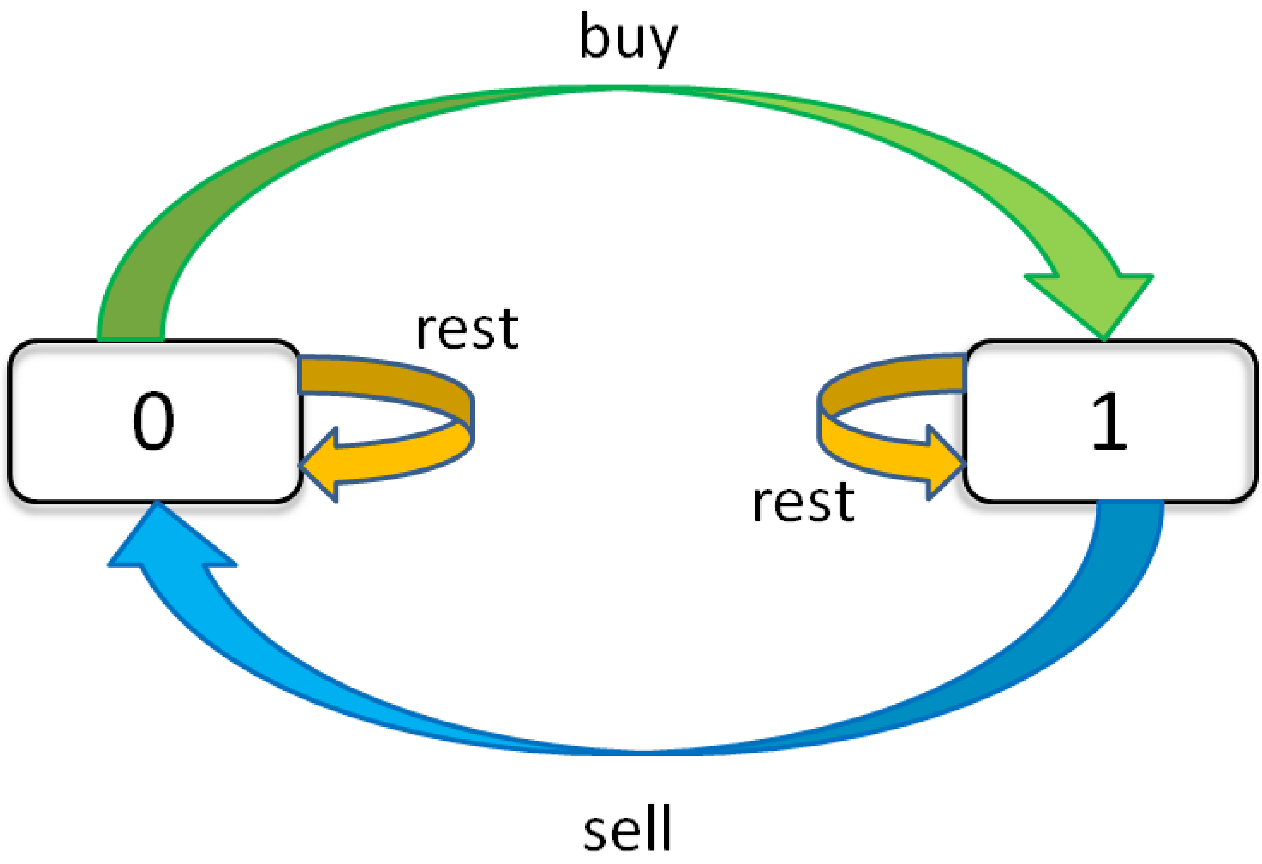
读者可能问为什么不是 `dp[n - 1][K][1]`？因为 `dp[n - 1][K][1]` 代表到最后一天手上还持有股票，`dp[n - 1][K][0]` 表示最后一天手上的股票已经卖出去了，很显然后者得到的利润一定大于前者。

记住如何解释「状态」，一旦你觉得哪里不好理解，把它翻译成自然语言就容易理解了。

二、状态转移框架

现在，我们完成了「状态」的穷举，我们开始思考每种「状态」有哪些「选择」，应该如何更新「状态」。

只看「持有状态」，可以画个状态转移图：



通过这个图可以很清楚地看到，每种状态（0 和 1）是如何转移而来的。根据这个图，我们来写一下状态转移方程：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
              max( 今天选择 rest,        今天选择 sell       )
```

解释：今天我没有持有股票，有两种可能，我从这两种可能中求最大利润：

1、我昨天就没有持有，且截至昨天最大交易次数限制为 k；然后我今天选择 rest，所以我今天还是没有持有，最大交易次数限制依然为 k。

2、我昨天持有股票，且截至昨天最大交易次数限制为 k；但是今天我 sell 了，所以我今天没有持有股票了，最大交易次数限制依然为 k。

```
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
              max( 今天选择 rest,         今天选择 buy         )
```

解释：今天我持有着股票，最大交易次数限制为 k，那么对于昨天来说，有两种可能，我从这两种可能中求最大利润：

1、我昨天就持有着股票，且截至昨天最大交易次数限制为 k；然后今天选择 rest，所以我今天还持有着股票，最大交易次数限制依然为 k。

2、我昨天本没有持有，且截至昨天最大交易次数限制为 `k - 1`；但今天我选择 `buy`，所以今天我就持有股票了，最大交易次数限制为 `k`。

> 这里着重提醒一下，**时刻牢记「状态」的定义**，状态 `k` 的定义并不是「已进行的交易次数」，而是「最大交易次数的上限限制」。如果确定今天进行一次交易，且要保证截至今天最大交易次数上限为 `k`，那么昨天的最大交易次数上限必须是 `k - 1`。

这个解释应该很清楚了，如果 `buy`，就要从利润中减去 `prices[i]`，如果 `sell`，就要给利润增加 `prices[i]`。今天的最大利润就是这两种可能选择中较大的那个。

注意 `k` 的限制，在选择 `buy` 的时候相当于开启了一次交易，那么对于昨天来说，交易次数的上限 `k` 应该减小 1。

> 修正：以前我以为在 `sell` 的时候给 `k` 减小 1 和在 `buy` 的时候给 `k` 减小 1 是等效的，但细心的读者向我提出质疑，经过深入思考我发现前者确实是错误的，因为交易是从 `buy` 开始，如果 `buy` 的选择不改变交易次数 `k` 的话，会出现交易次数超出限制的的错误。

现在，我们已经完成了动态规划中最困难的一步：状态转移方程。**如果之前的内容你都可以理解，那么你已经可以秒杀所有问题了，只要套这个框架就行了**。不过还差最后一点点，就是定义 base case，即最简单的情况。

```
dp[-1][...][0] = 0
解释：因为 i 是从 0 开始的，所以 i = -1 意味着还没有开始，这时候的利润当然是 0。

dp[-1][...][1] = -infinity
解释：还没开始的时候，是不可能持有股票的。
因为我们的算法要求一个最大值，所以初始值设为一个最小值，方便取最大值。

dp[...][0][0] = 0
解释：因为 k 是从 1 开始的，所以 k = 0 意味着根本不允许交易，这时候利润当然是 0。

dp[...][0][1] = -infinity
解释：不允许交易的情况下，是不可能持有股票的。
因为我们的算法要求一个最大值，所以初始值设为一个最小值，方便取最大值。
```

把上面的状态转移方程总结一下：

```
base case:
dp[-1][...][0] = dp[...][0][0] = 0
dp[-1][...][1] = dp[...][0][1] = -infinity

状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

读者可能会问，这个数组索引是 -1 怎么编程表示出来呢，负无穷怎么表示呢？这都是细节问题，有很多方法实现。现在完整的框架已经完成，下面开始具体化。

三、秒杀题目

第一题，先说力扣第 121 题「买卖股票的最佳时机」，相当于 `k = 1` 的情况：

121. 买卖股票的最佳时机 labuladong 题解 思路

难度 简单 2156

给定一个数组 `prices` ，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0` 。

示例 1：

```
输入：[7,1,5,3,6,4]
输出：5
解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。
```

示例 2：

```
输入：prices = [7,6,4,3,1]
输出：0
解释：在这种情况下，没有交易完成，所以最大利润为 0。
```

直接套状态转移方程，根据 base case，可以做一些化简：

```
dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])
dp[i][1][1] = max(dp[i-1][1][1], dp[i-1][0][0] - prices[i])
            = max(dp[i-1][1][1], -prices[i])
解释：k = 0 的 base case，所以 dp[i-1][0][0] = 0。

现在发现 k 都是 1，不会改变，即 k 对状态转移已经没有影响了。
可以进行进一步化简去掉所有 k：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], -prices[i])
```

直接写出代码：

```
int n = prices.length;
int[][] dp = new int[n][2];
for (int i = 0; i < n; i++) {
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
}
return dp[n - 1][0];
```

显然 `i = 0` 时 `i - 1` 是不合法的索引，这是因为我们没有对 `i` 的 base case 进行处理，可以这样给一个特化处理：

```
if (i - 1 == -1) {
    dp[i][0] = 0;
    // 根据状态转移方程可得：
    //   dp[i][0]
    // = max(dp[-1][0], dp[-1][1] + prices[i])
    // = max(0, -infinity + prices[i]) = 0

    dp[i][1] = -prices[i];
    // 根据状态转移方程可得：
    //   dp[i][1]
    // = max(dp[-1][1], dp[-1][0] - prices[i])
    // = max(-infinity, 0 - prices[i])
    // = -prices[i]
    continue;
}
```

第一题就解决了，但是这样处理 base case 很麻烦，而且注意一下状态转移方程，新状态只和相邻的一个状态有关，所以可以用前文 动态规划的降维打击：空间压缩技巧，不需要用整个 `dp` 数组，只需要一个变量储存相邻的那个状态就足够了，这样可以把空间复杂度降到 O(1):

```
// 原始版本
int maxProfit_k_1(int[] prices) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case
            dp[i][0] = 0;
            dp[i][1] = -prices[i];
            continue;
        }
        dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
    }
    return dp[n - 1][0];
}
```

```java
// 空间复杂度优化版本
int maxProfit_k_1(int[] prices) {
    int n = prices.length;
    // base case: dp[-1][0] = 0, dp[-1][1] = -infinity
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        // dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        // dp[i][1] = max(dp[i-1][1], -prices[i])
        dp_i_1 = Math.max(dp_i_1, -prices[i]);
    }
    return dp_i_0;
}
```

两种方式都是一样的，不过这种编程方法简洁很多，但是如果没有前面状态转移方程的引导，是肯定看不懂的。后续的题目，你可以对比一下如何把 dp 数组的空间优化掉。

第二题，看一下力扣第 122 题「买卖股票的最佳时机 II」，也就是 k 为正无穷的情况：

122. 买卖股票的最佳时机 II　labuladong 题解　思路

难度 中等　1572

给定一个数组 `prices` ，其中 `prices[i]` 表示股票第 `i` 天的价格。

在每一天，你可能会决定购买和/或出售股票。你在任何时候 **最多** 只能持有 **一股** 股票。你也可以购买它，然后在 **同一天** 出售。 返回你能获得的 **最大** 利润 。

示例 1:

```
输入: prices = [7,1,5,3,6,4]
输出: 7
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
     随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = 6-3 = 3 。
```

示例 2:

```
输入: prices = [1,2,3,4,5]
输出: 4
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
     注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
```

题目还专门强调可以在同一天出售，但我觉得这个条件纯属多余，如果当天买当天卖，那利润当然就是 0，这不是和没有进行交易是一样的吗？这道题的特点在于没有给出交易总数 k 的限制，也就相当于 k 为正无穷。

如果 k 为正无穷，那么就可以认为 k 和 k - 1 是一样的。可以这样改写框架：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
            = max(dp[i-1][k][1], dp[i-1][k][0] - prices[i])

我们发现数组中的 k 已经不会改变了，也就是说不需要记录 k 这个状态了：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])
```

直接翻译成代码：

```
// 原始版本
int maxProfit_k_inf(int[] prices) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case
            dp[i][0] = 0;
            dp[i][1] = -prices[i];
            continue;
        }
        dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i-1][1], dp[i-1][0] - prices[i]);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_k_inf(int[] prices) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, temp - prices[i]);
    }
    return dp_i_0;
}
```

第三题，看力扣第 309 题「最佳买卖股票时机含冷冻期」，也就是 k 为正无穷，但含有交易冷冻期的情况：

309. 最佳买卖股票时机含冷冻期 labuladong 题解 思路

难度 中等 1092

给定一个整数数组 `prices`，其中第 `prices[i]` 表示第 `i` 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票），但卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入: prices = [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]
```

和上一道题一样的，只不过每次 `sell` 之后要等一天才能继续交易，只要把这个特点融入上一题的状态转移方程即可：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])
解释：第 i 天选择 buy 的时候，要从 i-2 的状态转移，而不是 i-1 。
```

翻译成代码：

```java
// 原始版本
int maxProfit_with_cool(int[] prices) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case 1
            dp[i][0] = 0;
            dp[i][1] = -prices[i];
            continue;
        }
        if (i - 2 == -1) {
            // base case 2
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
            // i - 2 小于 0 时根据状态转移方程推出对应 base case
            dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
            //   dp[i][1]
            // = max(dp[i-1][1], dp[-1][0] - prices[i])
            // = max(dp[i-1][1], 0 - prices[i])
            // = max(dp[i-1][1], -prices[i])
```

```
                continue;
            }
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
            dp[i][1] = Math.max(dp[i-1][1], dp[i-2][0] - prices[i]);
        }
        return dp[n - 1][0];
    }

    // 空间复杂度优化版本
    int maxProfit_with_cool(int[] prices) {
        int n = prices.length;
        int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
        int dp_pre_0 = 0; // 代表 dp[i-2][0]
        for (int i = 0; i < n; i++) {
            int temp = dp_i_0;
            dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
            dp_i_1 = Math.max(dp_i_1, dp_pre_0 - prices[i]);
            dp_pre_0 = temp;
        }
        return dp_i_0;
    }
```

第四题，看力扣第 714 题「买卖股票的最佳时机含手续费」，也就是 k 为正无穷且考虑交易手续费的情况：

714. 买卖股票的最佳时机含手续费 labuladong 题解 思路

难度 中等 644

给定一个整数数组 `prices`，其中 `prices[i]` 表示第 `i` 天的股票价格 ；整数 `fee` 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

示例 1：

```
输入：prices = [1, 3, 2, 8, 4, 9], fee = 2
输出：8
解释：能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8
```

每次交易要支付手续费，只要把手续费从利润中减去即可，改写方程：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee)
解释：相当于买入股票的价格升高了。
在第一个式子里减也是一样的，相当于卖出股票的价格减小了。
```

> 如果直接把 `fee` 放在第一个式子里减，会有一些测试用例无法通过，错误原因是整型溢出而不是思路问题。一种解决方案是把代码中的 `int` 类型都改成 `long` 类型，避免 `int` 的整型溢出。

直接翻译成代码，注意状态转移方程改变后 base case 也要做出对应改变：

```
// 原始版本
int maxProfit_with_fee(int[] prices, int fee) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case
```

```
            dp[i][0] = 0;
            dp[i][1] = -prices[i] - fee;
            //   dp[i][1]
            // = max(dp[i - 1][1], dp[i - 1][0] - prices[i] - fee)
            // = max(dp[-1][1], dp[-1][0] - prices[i] - fee)
            // = max(-inf, 0 - prices[i] - fee)
            // = -prices[i] - fee
            continue;
        }
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
        dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i] - fee);
    }
    return dp[n - 1][0];
}

// 空间复杂度优化版本
int maxProfit_with_fee(int[] prices, int fee) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, temp - prices[i] - fee);
    }
    return dp_i_0;
}
```

第五题，看力扣第 123 题「买卖股票的最佳时机 III」，也就是 `k = 2` 的情况：

123. 买卖股票的最佳时机 III labuladong 题解 思路

难度 困难 1032

给定一个数组，它的第 `i` 个元素是一支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 **两笔** 交易。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入：prices = [3,3,5,0,0,3,1,4]
输出：6
解释：在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。
     随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天 （股票价格 = 4）的时候卖出，这笔交易所能获得利润 = 4-1 = 3 。
```

k = 2 和前面题目的情况稍微不同，因为上面的情况都和 k 的关系不太大：要么 k 是正无穷，状态转移和 k 没关系了；要么 k = 1，跟 k = 0 这个 base case 挨得近，最后也没有存在感。

这道题 k = 2 和后面要讲的 k 是任意正整数的情况中，对 k 的处理就凸显出来了，我们直接写代码，边写边分析原因。

```
原始的状态转移方程，没有可化简的地方
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

按照之前的代码，我们可能想当然这样写代码（错误的）：

```
int k = 2;
int[][][] dp = new int[n][k + 1][2];
for (int i = 0; i < n; i++) {
    if (i - 1 == -1) {
        // 处理 base case
        dp[i][k][0] = 0;
        dp[i][k][1] = -prices[i];
        continue;
    }
    dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
    dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
}
return dp[n - 1][k][0];
```

为什么错误？我这不是照着状态转移方程写的吗？

还记得前面总结的「穷举框架」吗？就是说我们必须穷举所有状态。其实我们之前的解法，都在穷举所有状态，只是之前的题目中 k 都被化简掉了。

比如说第一题，k = 1 时的代码框架：

```
int n = prices.length;
int[][] dp = new int[n][2];
for (int i = 0; i < n; i++) {
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
}
return dp[n - 1][0];
```

但当 k = 2 时，由于没有消掉 k 的影响，所以必须要对 k 进行穷举：

```
// 原始版本
int maxProfit_k_2(int[] prices) {
    int max_k = 2, n = prices.length;
```

```java
    int[][][] dp = new int[n][max_k + 1][2];
    for (int i = 0; i < n; i++) {
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) {
                // 处理 base case
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i];
                continue;
            }
            dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
            dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
        }
    }
    // 穷举了 n × max_k × 2 个状态，正确。
    return dp[n - 1][max_k][0];
}
```

> **PS：这里肯定会有读者疑惑，k 的 base case 是 0，按理说应该从 k = 1, k++ 这样穷举状态 k 才对？而且如果你真的这样从小到大遍历 k，提交发现也是可以的。**

这个疑问很正确，因为我们前文 动态规划答疑篇 有介绍 dp 数组的遍历顺序是怎么确定的，主要是根据 base case，以 base case 为起点，逐步向结果靠近。

但为什么我从大到小遍历 k 也可以正确提交呢？因为你注意看，dp[i][k][..] 不会依赖 dp[i][k - 1][..]，而是依赖 dp[i - 1][k - 1][..]，而 dp[i - 1][..][..]，都是已经计算出来的，所以不管你是 k = max_k, k--，还是 k = 1, k++，都是可以得出正确答案的。

那为什么我使用 k = max_k, k-- 的方式呢？因为这样符合语义：

你买股票，初始的「状态」是什么？应该是从第 0 天开始，而且还没有进行过买卖，所以最大交易次数限制 k 应该是 max_k；而随着「状态」的推移，你会进行交易，那么交易次数上限 k 应该不断减少，这样一想，k = max_k, k-- 的方式是比较合乎实际场景的。

当然，这里 k 取值范围比较小，所以也可以不用 for 循环，直接把 k = 1 和 2 的情况全部列举出来也可以：

```java
// 状态转移方程：
// dp[i][2][0] = max(dp[i-1][2][0], dp[i-1][2][1] + prices[i])
// dp[i][2][1] = max(dp[i-1][2][1], dp[i-1][1][0] - prices[i])
// dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])
// dp[i][1][1] = max(dp[i-1][1][1], -prices[i])

// 空间复杂度优化版本
int maxProfit_k_2(int[] prices) {
    // base case
    int dp_i10 = 0, dp_i11 = Integer.MIN_VALUE;
    int dp_i20 = 0, dp_i21 = Integer.MIN_VALUE;
    for (int price : prices) {
        dp_i20 = Math.max(dp_i20, dp_i21 + price);
        dp_i21 = Math.max(dp_i21, dp_i10 - price);
```

```
            dp_i10 = Math.max(dp_i10, dp_i11 + price);
            dp_i11 = Math.max(dp_i11, -price);
        }
        return dp_i20;
    }
```

有状态转移方程和含义明确的变量名指导，相信你很容易看懂。其实我们可以故弄玄虚，把上述四个变量换成 `a, b, c, d`。这样当别人看到你的代码时就会大惊失色，对你肃然起敬。

第六题，看力扣第 188 题「买卖股票的最佳时机 IV」，即 `k` 可以是题目给定的任何数的情况：

188. 买卖股票的最佳时机 IV

labuladong 题解 思路

难度 困难 667

给定一个整数数组 `prices` ，它的第 `i` 个元素 `prices[i]` 是一支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 **k** 笔交易。

注意： 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

```
输入：k = 2, prices = [3,2,6,5,0,3]
输出：7
解释：在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 = 6-2 = 4 。
     随后，在第 5 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出, 这笔交易所能获得利润 = 3-0 = 3 。
```

有了上一题 `k = 2` 的铺垫，这题应该和上一题的第一个解法没啥区别，你把上一题的 `k = 2` 换成题目输入的 `k` 就行了。

但试一下发现会出一个内存超限的错误，原来是传入的 `k` 值会非常大，`dp` 数组太大了。那么现在想想，交易次数 `k` 最多有多大呢？

一次交易由买入和卖出构成，至少需要两天。所以说有效的限制 `k` 应该不超过 `n/2`，如果超过，就没有约束作用了，相当于 `k` 没有限制的情况，而这种情况是之前解决过的。

所以我们可以直接把之前的代码重用：

```
int maxProfit_k_any(int max_k, int[] prices) {
    int n = prices.length;
```

```java
    if (n <= 0) {
        return 0;
    }
    if (max_k > n / 2) {
        // 复用之前交易次数 k 没有限制的情况
        return maxProfit_k_inf(prices);
    }

    // base case:
    // dp[-1][...][0] = dp[...][0][0] = 0
    // dp[-1][...][1] = dp[...][0][1] = -infinity
    int[][][] dp = new int[n][max_k + 1][2];
    // k = 0 时的 base case
    for (int i = 0; i < n; i++) {
        dp[i][0][1] = Integer.MIN_VALUE;
        dp[i][0][0] = 0;
    }

    for (int i = 0; i < n; i++)
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) {
                // 处理 i = -1 时的 base case
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i];
                continue;
            }
            dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] +
prices[i]);
            dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] -
prices[i]);
        }
    return dp[n - 1][max_k][0];
}
```

至此，6 道题目通过一个状态转移方程全部解决。

万法归一

如果你能看到这里，已经可以给你鼓掌了，初次理解如此复杂的动态规划问题想必消耗了你不少的脑细胞，不过这是值得的，股票系列问题已经属于动态规划问题中较困难的了，如果这些题你都能搞懂，试问，其他那些虾兵蟹将又何足道哉？

现在你已经过了九九八十一难中的前八十难，最后我还要再难为你一下，请你实现如下函数：

```java
int maxProfit_all_in_one(int max_k, int[] prices, int cooldown, int fee);
```

输入股票价格数组 `prices`，你最多进行 `max_k` 次交易，每次交易需要额外消耗 `fee` 的手续费，而且每次交易之后需要经过 `cooldown` 天的冷冻期才能进行下一次交易，请你计算并返回可以获得的最大利润。

怎么样，有没有被吓到？如果你直接给别人出一道这样的题目，估计对方要当场吐血，不过我们这样一步步做过来，你应该很容易发现这道题目就是之前我们探讨的几种情况的组合体嘛。

所以，我们只要把之前实现的几种代码掺和到一块，**在 base case 和状态转移方程中同时加上 `cooldown` 和 `fee` 的约束就行了**：

```
// 同时考虑交易次数的限制、冷冻期和手续费
int maxProfit_all_in_one(int max_k, int[] prices, int cooldown, int fee) {
    int n = prices.length;
    if (n <= 0) {
        return 0;
    }
    if (max_k > n / 2) {
        // 交易次数 k 没有限制的情况
        return maxProfit_k_inf(prices, cooldown, fee);
    }

    int[][][] dp = new int[n][max_k + 1][2];
    // k = 0 时的 base case
    for (int i = 0; i < n; i++) {
        dp[i][0][1] = Integer.MIN_VALUE;
        dp[i][0][0] = 0;
    }

    for (int i = 0; i < n; i++)
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) {
                // base case 1
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i] - fee;
                continue;
            }

            // 包含 cooldown 的 base case
            if (i - cooldown - 1 == -1) {
                // base case 2
                dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] +
prices[i]);
                // 别忘了减 fee
                dp[i][k][1] = Math.max(dp[i-1][k][1], -prices[i] - fee);
                continue;
            }
            dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] +
prices[i]);
            // 同时考虑 cooldown 和 fee
            dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-cooldown-1][0]
- prices[i] - fee);
        }
    return dp[n - 1][max_k][0];
}

// k 无限制，包含手续费和冷冻期
```

```
int maxProfit_k_inf(int[] prices, int cooldown, int fee) {
    int n = prices.length;
    int[][] dp = new int[n][2];
    for (int i = 0; i < n; i++) {
        if (i - 1 == -1) {
            // base case 1
            dp[i][0] = 0;
            dp[i][1] = -prices[i] - fee;
            continue;
        }

        // 包含 cooldown 的 base case
        if (i - cooldown - 1 == -1) {
            // base case 2
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
            // 别忘了减 fee
            dp[i][1] = Math.max(dp[i-1][1], -prices[i] - fee);
            continue;
        }
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
        // 同时考虑 cooldown 和 fee
        dp[i][1] = Math.max(dp[i - 1][1], dp[i - cooldown - 1][0] -
prices[i] - fee);
    }
    return dp[n - 1][0];
}
```

你可以用这个 `maxProfit_all_in_one` 函数去完成之前讲的 6 道题目，因为我们无法对 `dp` 数组进行优化，所以执行效率上不是最优的，但正确性上肯定是没有问题的。

最后总结一下吧，本文给大家讲了如何通过状态转移的方法解决复杂的问题，用一个状态转移方程秒杀了 6 道股票买卖问题，现在回头去看，其实也不算那么可怕对吧？

关键就在于列举出所有可能的「状态」，然后想想怎么穷举更新这些「状态」。一般用一个多维 `dp` 数组储存这些状态，从 base case 开始向后推进，推进到最后的状态，就是我们想要的答案。想想这个过程，你是不是有点理解「动态规划」这个名词的意义了呢？

具体到股票买卖问题，我们发现了三个状态，使用了一个三维数组，无非还是穷举 + 更新，不过我们可以说的高大上一点，这叫「三维 DP」，怕不怕？这个大实话一说，立刻显得你高人一等，名利双收有没有，所以给个在看/分享吧，鼓励一下我。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

团灭 LeetCode 打家劫舍问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

198. 打家劫舍（简单）

213. 打家劫舍II（中等）

337. 打家劫舍III（中等）

有读者私下问我 LeetCode 「打家劫舍」系列问题（英文版叫 House Robber）怎么做，我发现这一系列题目的点赞非常之高，是比较有代表性和技巧性的动态规划题目，今天就来聊聊这道题目。

打家劫舍系列总共有三道，难度设计非常合理，层层递进。第一道是比较标准的动态规划问题，而第二道融入了环形数组的条件，第三道更绝，把动态规划的自底向上和自顶向下解法和二叉树结合起来，我认为很有启发性。如果没做过的朋友，建议学习一下。

下面，我们从第一道开始分析。

————————————————

应合作方要求，本文不便在此发布，请扫码关注回复关键词「抢房子」查看：

动态规划之博弈问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

877. 石子游戏（中等）

上一篇文章 几道智力题 中讨论到一个有趣的「石头游戏」，通过题目的限制条件，这个游戏是先手必胜的。但是智力题终究是智力题，真正的算法问题肯定不会是投机取巧能搞定的。所以，本文就借石头游戏来讲讲「假设两个人都足够聪明，最后谁会获胜」这一类问题该如何用动态规划算法解决。

博弈类问题的套路都差不多，下文参考 这个 YouTube 视频 的思路讲解，其核心思路是在二维 dp 的基础上使用元组分别存储两个人的博弈结果。掌握了这个技巧以后，别人再问你什么俩海盗分宝石，俩人拿硬币的问题，你就告诉别人：我懒得想，直接给你写个算法算一下得了。

我们把力扣第 877 题「石头游戏」改的更具有一般性：

你和你的朋友面前有一排石头堆，用一个数组 `piles` 表示，`piles[i]` 表示第 `i` 堆石子有多少个。你们轮流拿石头，一次拿一堆，但是只能拿走最左边或者最右边的石头堆。所有石头被拿完后，谁拥有的石头多，谁获胜。

石头的堆数可以是任意正整数，石头的总数也可以是任意正整数，这样就能打破先手必胜的局面了。比如有三堆石头 `piles = [1, 100, 3]`，先手不管拿 1 还是 3，能够决定胜负的 100 都会被后手拿走，后手会获胜。

假设两人都很聪明，请你设计一个算法，返回先手和后手的最后得分（石头总数）之差。比如上面那个例子，先手能获得 4 分，后手会获得 100 分，你的算法应该返回 -96。

这样推广之后，这个问题算是一道 Hard 的动态规划问题了。**博弈问题的难点在于，两个人要轮流进行选择，而且都贼精明，应该如何编程表示这个过程呢？**

还是强调多次的套路，首先明确 `dp` 数组的含义，然后只要找到「状态」和「选择」，一切就水到渠成了。

一、定义 dp 数组的含义

定义 `dp` 数组的含义是很有技术含量的，同一问题可能有多种定义方法，不同的定义会引出不同的状态转移方程，不过只要逻辑没有问题，最终都能得到相同的答案。

我建议不要迷恋那些看起来很牛逼，代码很短小的奇技淫巧，最好是稳一点，采取可解释性最好，最容易推广的设计思路。本文就给出一种博弈问题的通用设计框架。

介绍 dp 数组的含义之前，我们先看一下 dp 数组最终的样子：

piles = [2, 8, 3, 5]

start \ end	0	1	2	3
0	(2, 0)	(8, 2)	(5, 8)	(13, 5)
1		(8, 0)	(8, 3)	(11, 5)
2			(3, 0)	(5, 3)
3				(5, 0)

下文讲解时，认为元组是包含 `first` 和 `second` 属性的一个类，而且为了节省篇幅，将这两个属性简写为 `fir` 和 `sec`。比如按上图的数据，我们说 `dp[1][3].fir = 11`，`dp[0][1].sec = 2`。

先回答几个读者可能提出的问题：

这个二维 dp table 中存储的是元组，怎么编程表示呢？这个 dp table 有一半根本没用上，怎么优化？很简单，都不要管，先把解题的思路想明白了再谈也不迟。

以下是对 dp 数组含义的解释：

`dp[i][j].fir = x` 表示，对于 `piles[i...j]` 这部分石头堆，先手能获得的最高分数为 `x`。

`dp[i][j].sec = y` 表示，对于 `piles[i...j]` 这部分石头堆，后手能获得的最高分数为 `y`。

举例理解一下，假设 `piles = [2, 8, 3, 5]`，索引从 0 开始，那么：

`dp[0][1].fir = 8` 意味着：面对石头堆 `[2, 8]`，先手最多能够获得 8 分；`dp[1][3].sec = 5` 意味着：面对石头堆 `[8, 3, 5]`，后手最多能够获得 5 分。

我们想求的答案是先手和后手最终分数之差，按照这个定义也就是 `dp[0][n-1].fir - dp[0][n-1].sec`，即面对整个 `piles`，先手的最优得分和后手的最优得分之差。

二、状态转移方程

写状态转移方程很简单，首先要找到所有「状态」和每个状态可以做的「选择」，然后择优。

根据前面对 dp 数组的定义，**状态显然有三个：开始的索引 i，结束的索引 j，当前轮到的人。**

```
dp[i][j][fir or sec]
其中:
0 <= i < piles.length
i <= j < piles.length
```

对于这个问题的每个状态，可以做的**选择有两个：选择最左边的那堆石头，或者选择最右边的那堆石头。** 我们可以这样穷举所有状态：

```
n = piles.length
for 0 <= i < n:
    for j <= i < n:
        for who in {fir, sec}:
            dp[i][j][who] = max(left, right)
```

上面的伪码是动态规划的一个大致的框架，这道题的难点在于，两人足够聪明，而且是交替进行选择的，也就是说先手的选择会对后手有影响，这怎么表达出来呢？

根据我们对 dp 数组的定义，很容易解决这个难点，**写出状态转移方程：**

```
dp[i][j].fir = max(piles[i] + dp[i+1][j].sec, piles[j] + dp[i][j-1].sec)
dp[i][j].fir = max(     选择最左边的石头堆     ,     选择最右边的石头堆      )
# 解释：我作为先手，面对 piles[i...j] 时，有两种选择：
# 要么我选择最左边的那一堆石头，然后面对 piles[i+1...j]
# 但是此时轮到对方，相当于我变成了后手；
# 要么我选择最右边的那一堆石头，然后面对 piles[i...j-1]
# 但是此时轮到对方，相当于我变成了后手。

if 先手选择左边:
    dp[i][j].sec = dp[i+1][j].fir
if 先手选择右边:
    dp[i][j].sec = dp[i][j-1].fir
# 解释：我作为后手，要等先手先选择，有两种情况：
# 如果先手选择了最左边那堆，给我剩下了 piles[i+1...j]
# 此时轮到我，我变成了先手；
# 如果先手选择了最右边那堆，给我剩下了 piles[i...j-1]
# 此时轮到我，我变成了先手。
```

根据 dp 数组的定义，我们也可以找出 **base case**，也就是最简单的情况：

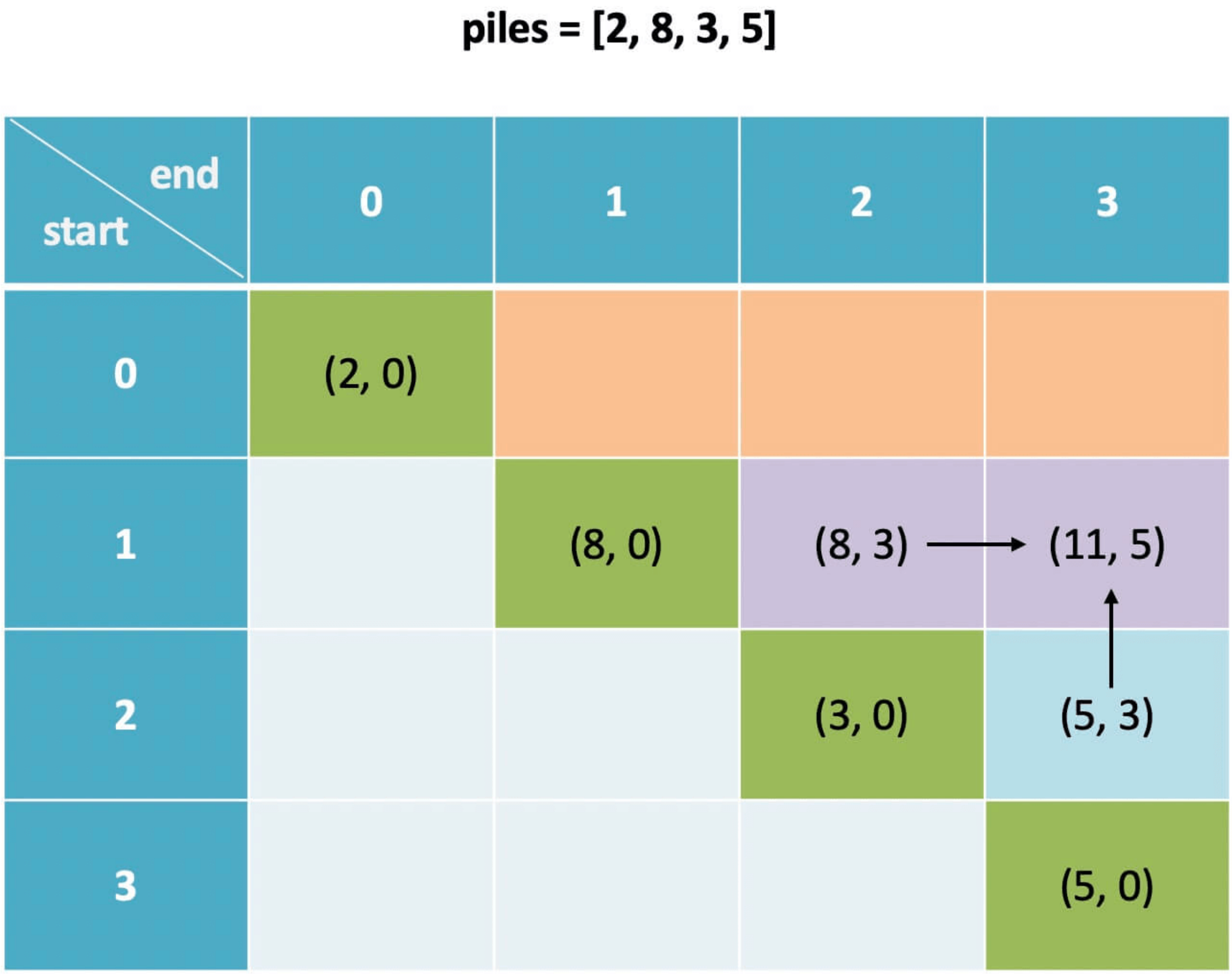
```
dp[i][j].fir = piles[i]
dp[i][j].sec = 0
其中 0 <= i == j < n
```

```
# 解释: i 和 j 相等就是说面前只有一堆石头 piles[i]
# 那么显然先手的得分为 piles[i]
# 后手没有石头拿了，得分为 0
```

piles = [2, 8, 3, 5]

start \ end	0	1	2	3
0	(2, 0)			
1		(8, 0)		
2			(3, 0)	
3				(5, 0)

这里需要注意一点，我们发现 base case 是斜着的，而且我们推算 `dp[i][j]` 时需要用到 `dp[i+1][j]` 和 `dp[i][j-1]`：



根据前文 动态规划答疑篇 判断 dp 数组遍历方向的原则，算法应该倒着遍历 dp 数组：

```
for (int i = n - 2; i >= 0; i--) {
    for (int j = i + 1; j < n; j++) {
        dp[i][j] = ...
    }
}
```

piles = [2, 8, 3, 5]

start \ end	0	1	2	3
0	(2, 0)	(8, 2)	(5, 8)	(13, 5)
1		(8, 0)	(8, 3)	(11, 5)
2			(3, 0)	(5, 3)
3				(5, 0)

三、代码实现

如何实现这个 fir 和 sec 元组呢，你可以用 python，自带元组类型；或者使用 C++ 的 pair 容器；或者用一个三维数组 `dp[n][n][2]`，最后一个维度就相当于元组；或者我们自己写一个 Pair 类：

```
class Pair {
    int fir, sec;
    Pair(int fir, int sec) {
        this.fir = fir;
        this.sec = sec;
    }
}
```

然后直接把我们的状态转移方程翻译成代码即可，注意我们要倒着遍历数组：

```
/* 返回游戏最后先手和后手的得分之差 */
int stoneGame(int[] piles) {
    int n = piles.length;
    // 初始化 dp 数组
    Pair[][] dp = new Pair[n][n];
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++)
```

```
                dp[i][j] = new Pair(0, 0);
        // 填入 base case
        for (int i = 0; i < n; i++) {
            dp[i][i].fir = piles[i];
            dp[i][i].sec = 0;
        }

        // 倒着遍历数组
        for (int i = n - 2; i >= 0; i--) {
            for (int j = i + 1; j < n; j++) {
                // 先手选择最左边或最右边的分数
                int left = piles[i] + dp[i+1][j].sec;
                int right = piles[j] + dp[i][j-1].sec;
                // 套用状态转移方程
                // 先手肯定会选择更大的结果，后手的选择随之改变
                if (left > right) {
                    dp[i][j].fir = left;
                    dp[i][j].sec = dp[i+1][j].fir;
                } else {
                    dp[i][j].fir = right;
                    dp[i][j].sec = dp[i][j-1].fir;
                }
            }
        }
        Pair res = dp[0][n-1];
        return res.fir - res.sec;
    }
```

动态规划解法，如果没有状态转移方程指导，绝对是一头雾水，但是根据前面的详细解释，读者应该可以清晰理解这一大段代码的含义。

而且，注意到计算 `dp[i][j]` 只依赖其左边和下边的元素，所以说肯定有优化空间，转换成一维 dp，想象一下把二维平面压扁，也就是投影到一维。但是，一维 `dp` 比较复杂，可解释性比较差，大家就不必浪费这个时间去理解了。

四、最后总结

本文给出了解决博弈问题的动态规划解法。博弈问题的前提一般都是在两个聪明人之间进行，编程描述这种游戏的一般方法是二维 dp 数组，数组中通过元组分别表示两人的最优决策。

之所以这样设计，是因为先手在做出选择之后，就成了后手，后手在对方做完选择后，就变成了先手。**这种角色转换使得我们可以重用之前的结果，典型的动态规划标志。**

读到这里的朋友应该能理解算法解决博弈问题的套路了。学习算法，一定要注重算法的模板框架，而不是一些看起来牛逼的思路，也不要奢求上来就写一个最优的解法。不要舍不得多用空间，不要过早尝试优化，不要惧怕多维数组。`dp` 数组就是存储信息避免重复计算的，随便用，直到咱满意为止。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



动态规划之最小路径和

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 [点这里](#)。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

[64. 最小路径和（中等）](#)

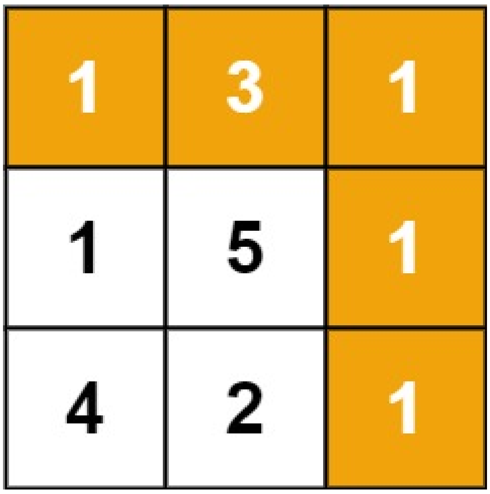
今天聊一道经典的动态规划题目，它是力扣第 64 题「[最小路径和](#)」，我来简单描述一下题目：

现在给你输入一个二维数组 `grid`，其中的元素都是**非负整数**，现在你站在左上角，**只能向右或者向下移动**，需要到达右下角。现在请你计算，经过的路径和最小是多少？

函数签名如下：

```java
int minPathSum(int[][] grid);
```

比如题目举的例子，输入如下的 `grid` 数组：

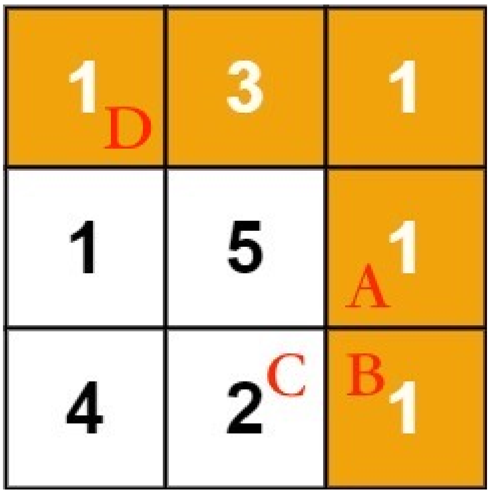


算法应该返回 7，最小路径和为 7，就是上图黄色的路径。

其实这道题难度不算大，但我们刷题群里很多朋友讨论，而且这个问题还有一些难度比较大的变体，所以讲一下这种问题的通用思路。

一般来说，让你在二维矩阵中求最优化问题（最大值或者最小值），肯定需要递归 + 备忘录，也就是动态规划技巧。

就拿题目举的例子来说，我给图中的几个格子编个号方便描述：



我们想计算从起点 D 到达 B 的最小路径和，那你说怎么才能到达 B 呢？

题目说了只能向右或者向下走，所以只有从 A 或者 C 走到 B。

那么算法怎么知道从 A 走到 B 才能使路径和最小，而不是从 C 走到 B 呢？

难道是因为位置 A 的元素大小是 1，位置 C 的元素是 2，1 小于 2，所以一定要从 A 走到 B 才能使路径和最小吗？

其实不是的，**真正的原因是，从 D 走到 A 的最小路径和是 6，而从 D 走到 C 的最小路径和是 8，6 小于 8，所以一定要从 A 走到 B 才能使路径和最小。**

换句话说，我们把「从 D 走到 B 的最小路径和」这个问题转化成了「从 D 走到 A 的最小路径和」和 「从 D 走到 C 的最小路径和」这两个问题。

理解了上面的分析，这不就是状态转移方程吗？所以这个问题肯定会用到动态规划技巧来解决。

比如我们定义如下一个 dp 函数：

```
int dp(int[][] grid, int i, int j);
```

这个 dp 函数的定义如下：

从左上角位置 `(0, 0)` 走到位置 `(i, j)` 的最小路径和为 `dp(grid, i, j)`。

根据这个定义，我们想求的最小路径和就可以通过调用这个 dp 函数计算出来：

```
int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    // 计算从左上角走到右下角的最小路径和
    return dp(grid, m - 1, n - 1);
}
```

再根据刚才的分析，很容易发现，`dp(grid, i, j)` 的值取决于 `dp(grid, i - 1, j)` 和 `dp(grid, i, j - 1)` 返回的值。

我们可以直接写代码了：

```
int dp(int[][] grid, int i, int j) {
    // base case
    if (i == 0 && j == 0) {
        return grid[0][0];
    }
    // 如果索引出界，返回一个很大的值，
    // 保证在取 min 的时候不会被取到
    if (i < 0 || j < 0) {
        return Integer.MAX_VALUE;
    }

    // 左边和上面的最小路径和加上 grid[i][j]
    // 就是到达 (i, j) 的最小路径和
    return Math.min(
            dp(grid, i - 1, j),
            dp(grid, i, j - 1)
        ) + grid[i][j];
}
```

上述代码逻辑已经完整了，接下来就分析一下，这个递归算法是否存在重叠子问题？是否需要用备忘录优化一下执行效率？

前文多次说过判断重叠子问题的技巧，首先抽象出上述代码的递归框架：

```
int dp(int i, int j) {
    dp(i - 1, j); // #1
    dp(i, j - 1); // #2
}
```

如果我想从 `dp(i, j)` 递归到 `dp(i-1, j-1)`，有几种不同的递归调用路径？

可以是 `dp(i, j) -> #1 -> #2` 或者 `dp(i, j) -> #2 -> #1`，不止一种，说明 `dp(i-1, j-1)` 会被多次计算，所以一定存在重叠子问题。

那么我们可以使用备忘录技巧进行优化：

```
int[][] memo;

int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    // 构造备忘录，初始值全部设为 -1
    memo = new int[m][n];
```

```java
        for (int[] row : memo)
            Arrays.fill(row, -1);

        return dp(grid, m - 1, n - 1);
    }

    int dp(int[][] grid, int i, int j) {
        // base case
        if (i == 0 && j == 0) {
            return grid[0][0];
        }
        if (i < 0 || j < 0) {
            return Integer.MAX_VALUE;
        }
        // 避免重复计算
        if (memo[i][j] != -1) {
            return memo[i][j];
        }
        // 将计算结果记入备忘录
        memo[i][j] = Math.min(
            dp(grid, i - 1, j),
            dp(grid, i, j - 1)
        ) + grid[i][j];

        return memo[i][j];
    }
```

至此，本题就算是解决了，时间复杂度和空间复杂度都是 `O(MN)`，标准的自顶向下动态规划解法。

有的读者可能问，能不能用自底向上的迭代解法来做这道题呢？完全可以的。

首先，类似刚才的 `dp` 函数，我们需要一个二维 `dp` 数组，定义如下：

从左上角位置 `(0, 0)` 走到位置 `(i, j)` 的最小路径和为 `dp[i][j]`。

状态转移方程当然不会变的，`dp[i][j]` 依然取决于 `dp[i-1][j]` 和 `dp[i][j-1]`，直接看代码吧：

```java
int minPathSum(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        int[][] dp = new int[m][n];

        /**** base case ****/
        dp[0][0] = grid[0][0];

        for (int i = 1; i < m; i++)
            dp[i][0] = dp[i - 1][0] + grid[i][0];

        for (int j = 1; j < n; j++)
            dp[0][j] = dp[0][j - 1] + grid[0][j];
        /*******************/
```

```
        // 状态转移
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = Math.min(
                    dp[i - 1][j],
                    dp[i][j - 1]
                ) + grid[i][j];
            }
        }

        return dp[m - 1][n - 1];
    }
```

这个解法的 base case 看起来和递归解法略有不同，但实际上是一样的。

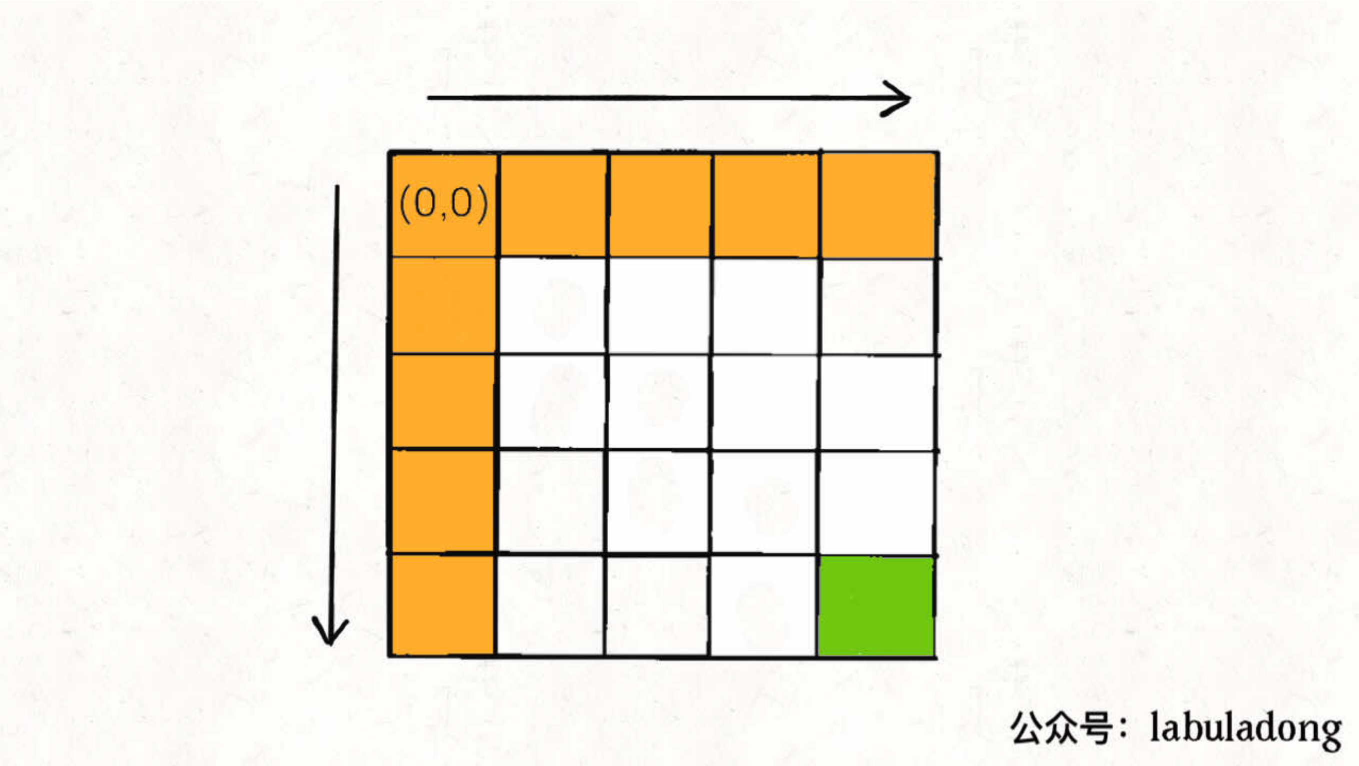
因为状态转移为下面这段代码：

```
dp[i][j] = Math.min(
    dp[i - 1][j],
    dp[i][j - 1]
) + grid[i][j];
```

那如果 `i` 或者 `j` 等于 0 的时候，就会出现索引越界的错误。

所以我们需要提前计算出 `dp[0][..]` 和 `dp[..][0]`，然后让 `i` 和 `j` 的值从 1 开始迭代。

`dp[0][..]` 和 `dp[..][0]` 的值怎么算呢？其实很简单，第一行和第一列的路径和只有下面这一种情况嘛：



那么按照 `dp` 数组的定义，`dp[i][0] = sum(grid[0..i][0]), dp[0][j] = sum(grid[0][0..j])`，也就是如下代码：

```
/**** base case ****/
dp[0][0] = grid[0][0];

for (int i = 1; i < m; i++)
    dp[i][0] = dp[i - 1][0] + grid[i][0];

for (int j = 1; j < n; j++)
    dp[0][j] = dp[0][j - 1] + grid[0][j];
/*******************/
```

到这里，自底向上的迭代解法也搞定了，那有的读者可能又要问了，能不能优化一下算法的空间复杂度呢？

前文 动态规划的降维打击：空间压缩 说过降低 `dp` 数组的技巧，这里也是适用的，不过略微复杂些，本文由于篇幅所限就不写了，有兴趣的读者可以自己尝试一下。

本文到此结束，下篇文章写一道进阶题目，更加巧妙和有趣，敬请期待~

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



经典动态规划：高楼扔鸡蛋

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

887. 鸡蛋掉落（困难）

今天要聊一个很经典的算法问题，若干层楼，若干个鸡蛋，让你算出最少的尝试次数，找到鸡蛋恰好摔不碎的那层楼。国内大厂以及谷歌脸书面试都经常考察这道题，只不过他们觉得扔鸡蛋太浪费，改成扔杯子，扔破碗什么的。

具体的问题等会再说，但是这道题的解法技巧很多，光动态规划就好几种效率不同的思路，最后还有一种极其高效数学解法。秉承咱们号一贯的作风，拒绝奇技淫巧，拒绝过于诡异的技巧，因为这些技巧无法举一反三，学了也不划算。

下面就来用我们一直强调的动态规划通用思路来研究一下这道题。

一、解析题目

这是力扣第 887 题「鸡蛋掉落」，我描述一下题目：

你面前有一栋从 1 到 `N` 共 `N` 层的楼，然后给你 `K` 个鸡蛋（`K` 至少为 1）。现在确定这栋楼存在楼层 `0 <= F <= N`，在这层楼将鸡蛋扔下去，鸡蛋**恰好没摔碎**（高于 `F` 的楼层都会碎，低于 `F` 的楼层都不会碎）。现在问你，**最坏**情况下，你**至少**要扔几次鸡蛋，才能**确定**这个楼层 `F` 呢?

也就是让你找摔不碎鸡蛋的最高楼层 `F`，但什么叫「最坏情况」下「至少」要扔几次呢？我们分别举个例子就明白了。

比方说**现在先不管鸡蛋个数的限制**，有 7 层楼，你怎么去找鸡蛋恰好摔碎的那层楼?

最原始的方式就是线性扫描：我先在 1 楼扔一下，没碎，我再去 2 楼扔一下，没碎，我再去 3 楼......

以这种策略，**最坏**情况应该就是我试到第 7 层鸡蛋也没碎（`F = 7`），也就是我扔了 7 次鸡蛋。

先在你应该理解什么叫做「最坏情况」下了，**鸡蛋破碎一定发生在搜索区间穷尽时**，不会说你在第 1 层摔一下鸡蛋就碎了，这是你运气好，不是最坏情况。

现在再来理解一下什么叫「至少」要扔几次。依然不考虑鸡蛋个数限制，同样是 7 层楼，我们可以优化策略。

最好的策略是使用二分查找思路，我先去第 `(1 + 7) / 2 = 4` 层扔一下：

如果碎了说明 `F` 小于 4，我就去第 `(1 + 3) / 2 = 2` 层试……

如果没碎说明 `F` 大于等于 4，我就去第 `(5 + 7) / 2 = 6` 层试……

以这种策略，**最坏**情况应该是试到第 7 层鸡蛋还没碎（`F = 7`），或者鸡蛋一直碎到第 1 层（`F = 0`）。然而无论那种最坏情况，只需要试 `log7` 向上取整等于 3 次，比刚才尝试 7 次要少，这就是所谓的**至少**要扔几次。

> PS：这有点像 Big O 表示法计算算法的复杂度。

实际上，如果不限制鸡蛋个数的话，二分思路显然可以得到最少尝试的次数，但问题是，**现在给你了鸡蛋个数的限制 `K`，直接使用二分思路就不行了**。

比如说只给你 1 个鸡蛋，7 层楼，你敢用二分吗？你直接去第 4 层扔一下，如果鸡蛋没碎还好，但如果碎了你就没有鸡蛋继续测试了，无法确定鸡蛋恰好摔不碎的楼层 `F` 了。这种情况下只能用线性扫描的方法，算法返回结果应该是 7。

有的读者也许会有这种想法：二分查找排除楼层的速度无疑是最快的，那干脆先用二分查找，等到只剩 1 个鸡蛋的时候再执行线性扫描，这样得到的结果是不是就是最少的扔鸡蛋次数呢？

很遗憾，并不是，比如说把楼层变高一些，100 层，给你 2 个鸡蛋，你在 50 层扔一下，碎了，那就只能线性扫描 1～49 层了，最坏情况下要扔 50 次。

如果不要「二分」，变成「五分」「十分」都会大幅减少最坏情况下的尝试次数。比方说第一个鸡蛋每隔十层楼扔，在哪里碎了第二个鸡蛋一个个线性扫描，总共不会超过 20 次。

最优解其实是 14 次。最优策略非常多，而且并没有什么规律可言。

说了这么多废话，就是确保大家理解了题目的意思，而且认识到这个题目确实复杂，就连我们手算都不容易，如何用算法解决呢？

二、思路分析

————————————————

应合作方要求，本文不便在此发布，请扫码关注回复关键词「鸡蛋」查看：

动态规划帮我通关了《魔塔》

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

174. 地下城游戏（困难）

「魔塔」是一款经典的地牢类游戏，碰怪物要掉血，吃血瓶能加血，你要收集钥匙，一层一层上楼，最后救出美丽的公主。

现在手机上仍然可以玩这个游戏：


106
11
3
1
第8层
1345
20
30
无
无

嗯，相信这款游戏承包了不少人的童年回忆，记得小时候，一个人拿着游戏机玩，两三个人围在左右指手画脚，这导致玩游戏的人体验极差，而左右的人异常快乐 😂

力扣第 174 题「地下城游戏」是一道类似的题目，我简单描述一下：

输入一个存储着整数的二维数组 `grid`，如果 `grid[i][j] > 0`，说明这个格子装着血瓶，经过它可以增加对应的生命值；如果 `grid[i][j] == 0`，则这是一个空格子，经过它不会发生任何事情；如果 `grid[i][j] < 0`，说明这个格子有怪物，经过它会损失对应的生命值。

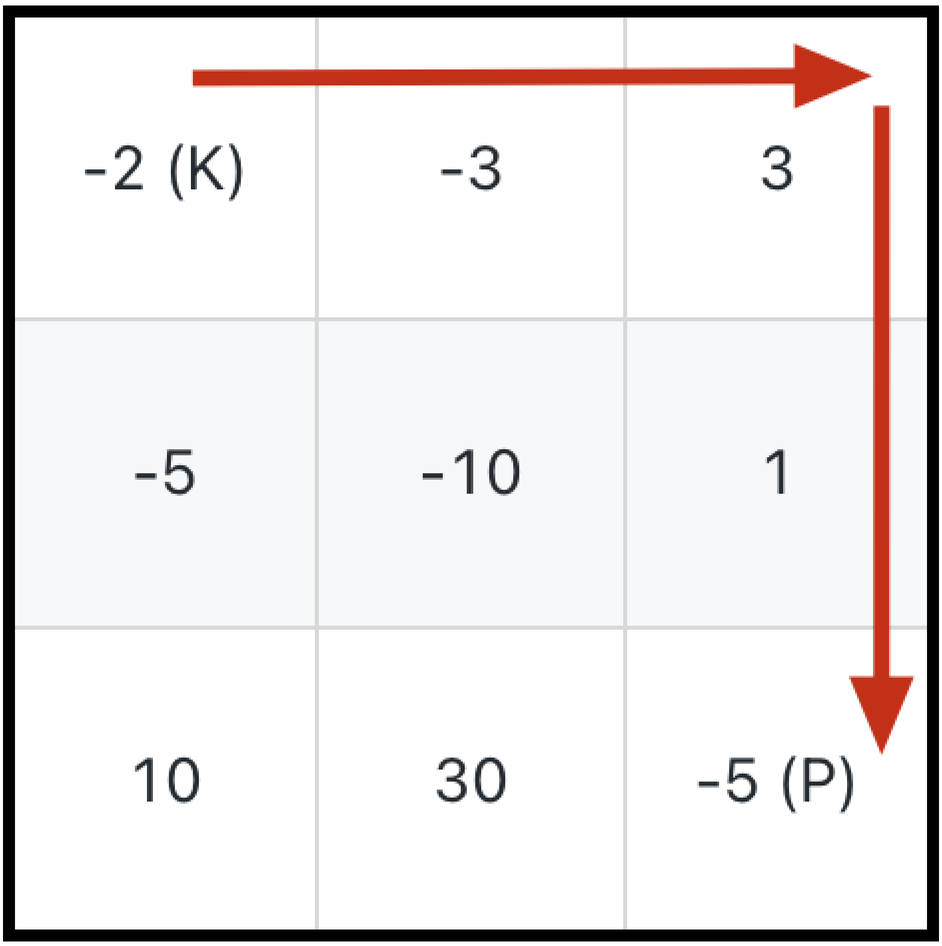
现在你是一名骑士，将会出现在最上角，公主被困在最右下角，你只能向右和向下移动，请问你初始至少需要多少生命值才能成功救出公主？

换句话说，就是问你至少需要多少初始生命值，能够让骑士从最左上角移动到最右下角，且任何时候生命值都要大于 0。

函数签名如下：

```
int calculateMinimumHP(int[][] grid);
```

比如题目给我们举的例子，输入如下一个二维数组 `grid`，用 `K` 表示骑士，用 `P` 表示公主：



算法应该返回 7，也就是说骑士的初始生命值**至少**为 7 时才能成功救出公主，行进路线如图中的箭头所示。

上篇文章 最小路径和 写过类似的问题，问你从左上角到右下角的最小路径和是多少。

我们做算法题一定要尝试举一反三，感觉今天这道题和最小路径和有点关系对吧？

想要最小化骑士的初始生命值，是不是意味着要最大化骑士行进路线上的血瓶？是不是相当于求「最大路径和」？是不是可以直接套用计算「最小路径和」的思路？

但是稍加思考，发现这个推论并不成立，吃到最多的血瓶，并不一定就能获得最小的初始生命值。

比如如下这种情况，如果想要吃到最多的血瓶获得「最大路径和」，应该按照下图箭头所示的路径，初始生命值需要 11：



但也很容易看到，正确的答案应该是下图箭头所示的路径，初始生命值只需要 1：



所以，关键不在于吃最多的血瓶，而是在于如何损失最少的生命值。

这类求最值的问题，肯定要借助动态规划技巧，要合理设计 dp 数组/函数的定义。类比前文 最小路径和问题，dp 函数签名肯定长这样：

```
int dp(int[][] grid, int i, int j);
```

但是这道题对 dp 函数的定义比较有意思，按照常理，这个 dp 函数的定义应该是：

从左上角（grid[0][0]）走到 grid[i][j] 至少需要 dp(grid, i, j) 的生命值。

这样定义的话，base case 就是 i, j 都等于 0 的时候，我们可以这样写代码：

```
int calculateMinimumHP(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    // 我们想计算左上角到右下角所需的最小生命值
    return dp(grid, m - 1, n - 1);
}

int dp(int[][] grid, int i, int j) {
    // base case
    if (i == 0 && j == 0) {
```

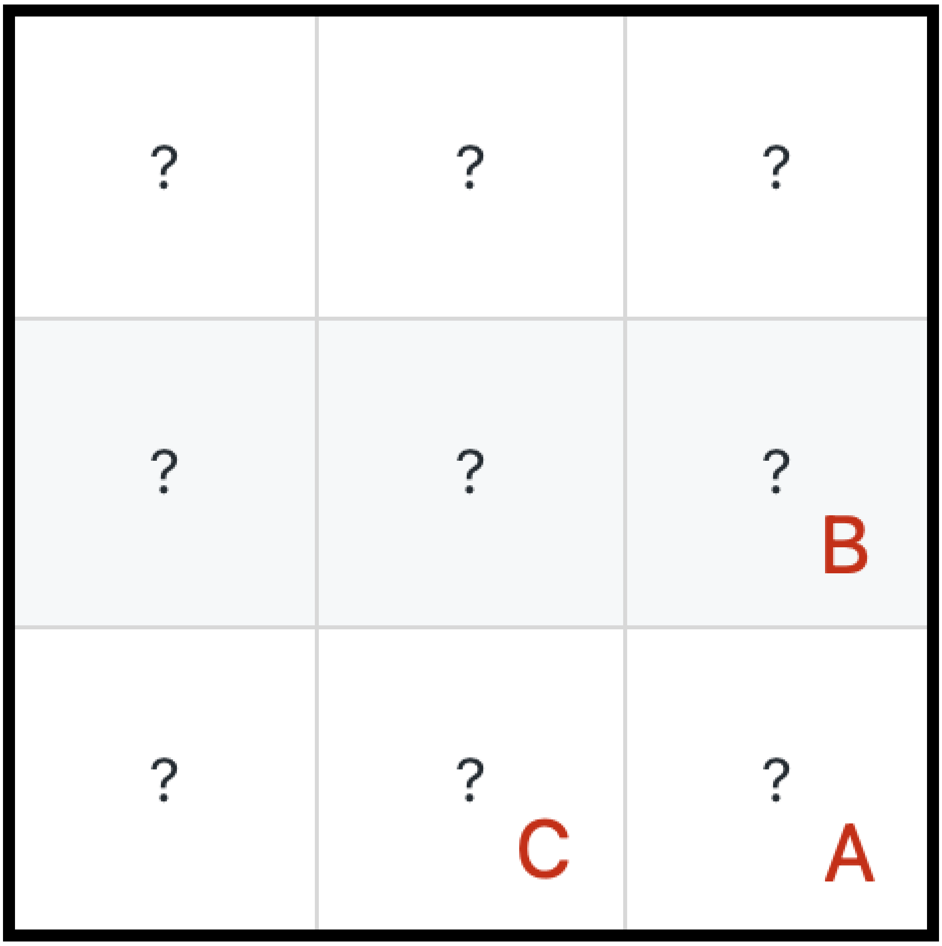
```
            // 保证骑士落地不死就行了
            return gird[i][j] > 0 ? 1 : -grid[i][j] + 1;
        }
        ...
}
```

> **PS：为了简洁，之后 `dp(grid, i, j)` 就简写为 `dp(i, j)`，大家理解就好。**

接下来我们需要找状态转移了，还记得如何找状态转移方程吗？我们这样定义 `dp` 函数能否正确进行状态转移呢？

我们希望 `dp(i, j)` 能够通过 `dp(i-1, j)` 和 `dp(i, j-1)` 推导出来，这样就能不断逼近 base case，也就能够正确进行状态转移。

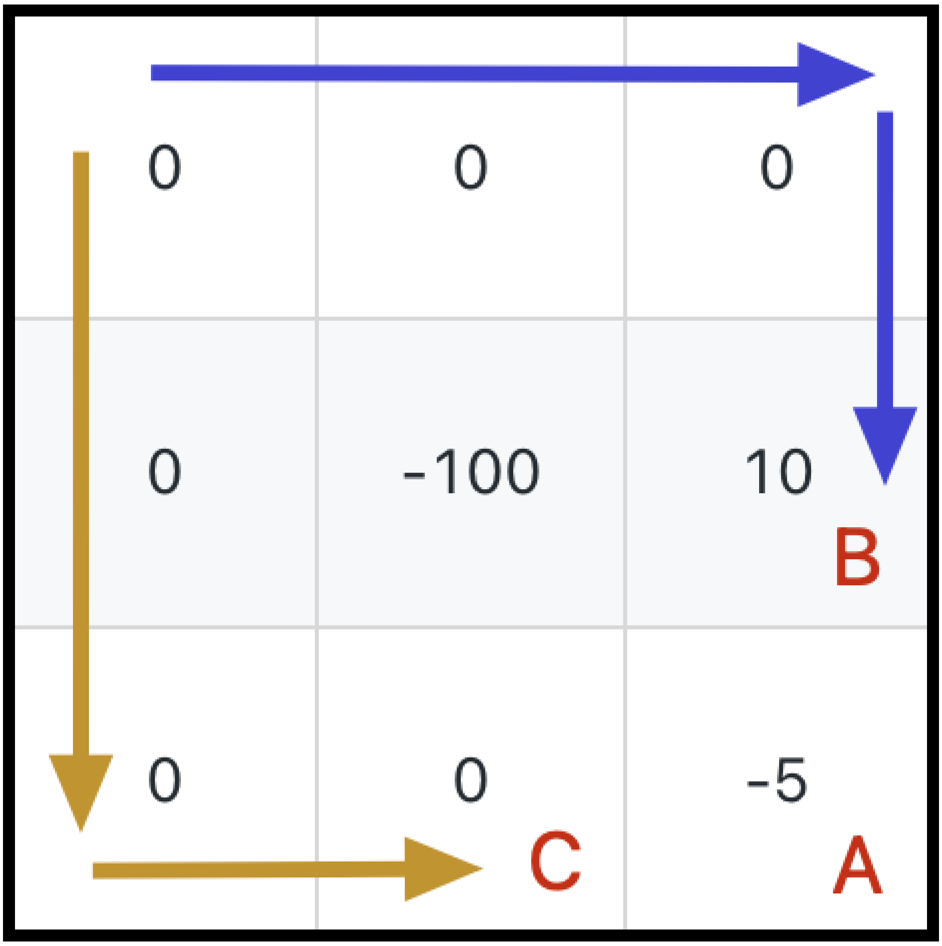
具体来说，「到达 `A` 的最小生命值」应该能够由「到达 `B` 的最小生命值」和「到达 `C` 的最小生命值」推导出来：



但问题是，能推出来么？实际上是不能的。

因为按照 `dp` 函数的定义，你只知道「能够从左上角到达 `B` 的最小生命值」，但并不知道「到达 `B` 时的生命值」。

「到达 `B` 时的生命值」是进行状态转移的必要参考，我给你举个例子你就明白了，假设下图这种情况：



你说这种情况下，骑士救公主的最优路线是什么？

显然是按照图中蓝色的线走到 B，最后走到 A 对吧，这样初始血量只需要 1 就可以；如果走黄色箭头这条路，先走到 C 然后走到 A，初始血量至少需要 6。

为什么会这样呢？骑士走到 B 和 C 的最少初始血量都是 1，为什么最后是从 B 走到 A，而不是从 C 走到 A 呢？

因为骑士走到 B 的时候生命值为 11，而走到 C 的时候生命值依然是 1。

如果骑士执意要通过 C 走到 A，那么初始血量必须加到 6 点才行；而如果通过 B 走到 A，初始血量为 1 就够了，因为路上吃到血瓶了，生命值足够抗 A 上面怪物的伤害。

这下应该说的很清楚了，再回顾我们对 dp 函数的定义，上图的情况，算法只知道 dp(1, 2) = dp(2, 1) = 1，都是一样的，怎么做出正确的决策，计算出 dp(2, 2) 呢？

所以说，我们之前对 dp 数组的定义是错误的，信息量不足，算法无法做出正确的状态转移。

正确的做法需要反向思考，依然是如下的 dp 函数：

```
int dp(int[][] grid, int i, int j);
```

但是我们要修改 dp 函数的定义：

从 `grid[i][j]` 到达终点（右下角）所需的最少生命值是 `dp(grid, i, j)`。

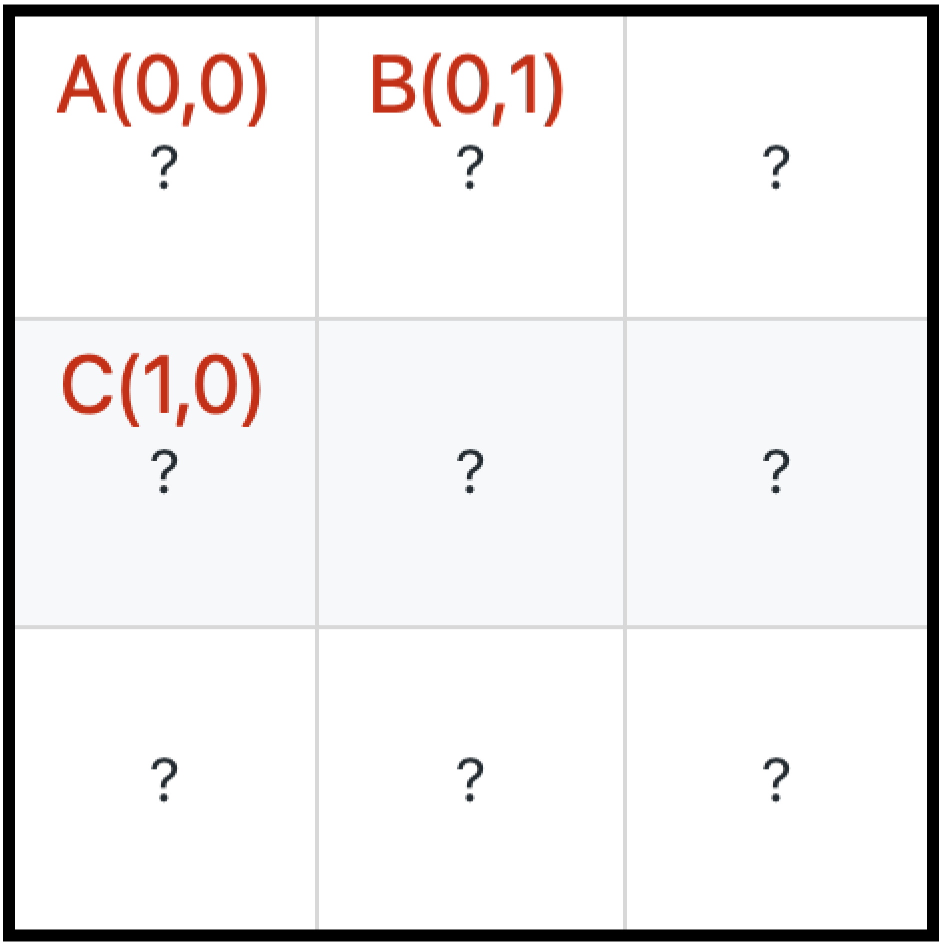
那么可以这样写代码：

```
int calculateMinimumHP(int[][] grid) {
    // 我们想计算左上角到右下角所需的最小生命值
    return dp(grid, 0, 0);
}

int dp(int[][] grid, int i, int j) {
    int m = grid.length;
    int n = grid[0].length;
    // base case
    if (i == m - 1 && j == n - 1) {
        return grid[i][j] >= 0 ? 1 : -grid[i][j] + 1;
    }
    ...
}
```

根据新的 `dp` 函数定义和 base case，我们想求 `dp(0, 0)`，那就应该试图通过 `dp(i, j+1)` 和 `dp(i+1, j)` 推导出 `dp(i, j)`，这样才能不断逼近 base case，正确进行状态转移。

具体来说，「从 `A` 到达右下角的最少生命值」应该由「从 `B` 到达右下角的最少生命值」和「从 `C` 到达右下角的最少生命值」推导出来：



能不能推导出来呢？这次是可以的，假设 dp(0, 1) = 5, dp(1, 0) = 4，那么可以肯定要从 A 走向 C，因为 4 小于 5 嘛。

那么怎么推出 dp(0, 0) 是多少呢？

假设 A 的值为 1，既然知道下一步要往 C 走，且 dp(1, 0) = 4 意味着走到 grid[1][0] 的时候至少要有 4 点生命值，那么就可以确定骑士出现在 A 点时需要 4 - 1 = 3 点初始生命值，对吧。

那如果 A 的值为 10，落地就能捡到一个大血瓶，超出了后续需求，4 - 10 = -6 意味着骑士的初始生命值为负数，这显然不可以，骑士的生命值小于 1 就挂了，所以这种情况下骑士的初始生命值应该是 1。

综上，状态转移方程已经推出来了：

```
int res = min(
    dp(i + 1, j),
    dp(i, j + 1)
) - grid[i][j];

dp(i, j) = res <= 0 ? 1 : res;
```

根据这个核心逻辑，加一个备忘录消除重叠子问题，就可以直接写出最终的代码了：

```java
/* 主函数 */
int calculateMinimumHP(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    // 备忘录中都初始化为 -1
    memo = new int[m][n];
    for (int[] row : memo) {
        Arrays.fill(row, -1);
    }

    return dp(grid, 0, 0);
}

// 备忘录，消除重叠子问题
int[][] memo;

/* 定义：从 (i, j) 到达右下角，需要的初始血量至少是多少 */
int dp(int[][] grid, int i, int j) {
    int m = grid.length;
    int n = grid[0].length;
    // base case
    if (i == m - 1 && j == n - 1) {
        return grid[i][j] >= 0 ? 1 : -grid[i][j] + 1;
    }
    if (i == m || j == n) {
        return Integer.MAX_VALUE;
    }
    // 避免重复计算
    if (memo[i][j] != -1) {
        return memo[i][j];
    }
    // 状态转移逻辑
    int res = Math.min(
            dp(grid, i, j + 1),
            dp(grid, i + 1, j)
        ) - grid[i][j];
    // 骑士的生命值至少为 1
    memo[i][j] = res <= 0 ? 1 : res;

    return memo[i][j];
}
```

这就是自顶向下带备忘录的动态规划解法，参考前文 动态规划套路详解 很容易就可以改写成 dp 数组的迭代解法，这里就不写了，读者可以尝试自己写一写。

这道题的核心是定义 dp 函数，找到正确的状态转移方程，从而计算出正确的答案。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



动态规划帮我通关了《辐射4》

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

514. 自由之路（困难）

本文的封面图是一款叫做《辐射4》的游戏中的一个任务剧情画面：



这个可以转动的圆盘类似是一个密码机关，中间偏上的位置有个红色的指针看到没，你只要转动圆盘可以让指针指向不同的字母，然后再按下中间的按钮就可以输入指针指向的字母。

只要转动圆环，让指针依次指向 R、A、I、L、R、O、A、D 并依次按下按钮，就可以触发机关，打开旁边的门。

至于密码为什么是这几个字母，在游戏中的剧情有暗示，这里就不多说了。

那么这个游戏场景和动态规划有什么关系呢？

我们来没事儿找事儿地想一想，拨动圆盘输入这些字母还挺麻烦的，按照什么顺序才能使得拨动圆盘所需的操作次数最少呢？

拨动圆盘的不同方法所需的操作次数肯定是不同的。

比如说你想把一个字母对准到指针上，你可以顺时针转圆盘，也可以逆时针转圆盘；而且某些字母可能不止出现一次，比如上图中大写字母 O 就在圆盘的不同位置出现了三次，你到时候应该拨哪个 O 才能使得整体的操作次数最少呢？
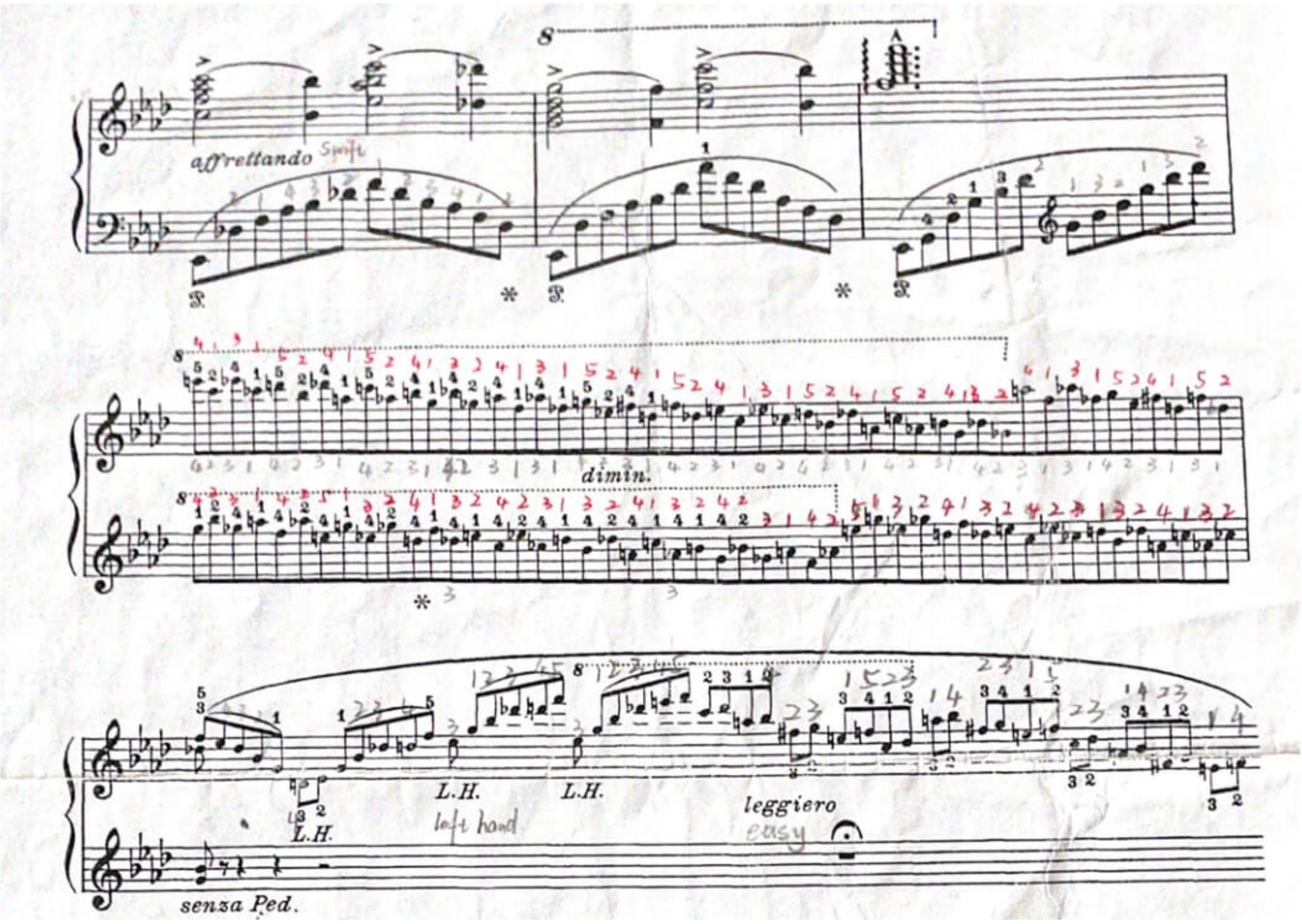
我们之前也多次说过，遇到求最值的问题，基本都是由动态规划算法来解决，因为动态规划本身就是运筹优化算法的一种嘛。

力扣上就有一道这个转盘游戏的算法题，难度还是 Hard，但我当时看了一眼就做出来了，因为我以前思考过生活中一个非常有意思的例子可以类比到这个问题，下面来简单介绍一下。

关注了我的视频号的朋友，知道我弹过李斯特和肖邦的几首钢琴曲，但是没练过钢琴的读者可能不知道，练习钢琴曲谱是需要提前确定「指法」的。

五线谱的音符七上八下的，两个手的手指必须互相配合，也就是说你必须确定好每个音符用哪只手的哪个手指来弹奏，写到谱子上。

比如说我很喜欢的一首曲子叫做《爱之梦》，这是我的谱子：



音符上的数字 1 代表用大拇指，2 代表用食指，以此类推。按照确定下来的指法不断练习，形成肌肉记忆，就算是练会一首曲子了。

指法这东西因人而异，比如手大的人可以让中指跨到大拇指的左边，手小的人可能就有些别扭，那同一段谱子对应的指法可能就不一样。

那么问题来了，我应该如何设计指法，才能最小化手指切换的「别扭程度」，也就是最大化演奏的流畅度呢？

这里我就借助了动态规划算法技巧：手指的切换不就是状态的转移么？参考前文 动态规划套路详解，只要明确「状态」和「选择」就可以解决这个问题。

状态是什么？状态就是「下一个需要弹奏的音符」和「当前的手的状态」。

下一个需要弹奏的音符，无非就是钢琴上 88 个琴键中的一个；手的状态也很简单，五个手指头，每个手指头要么按下去了要么没按下去，2 的 5 次方 32 种情况，5 个二进制位就可以表示。

选择是什么？选择就是「下一个音符应该由哪个手指头来弹」，无非就是穷举五个手指头。

当然，结合当前手的状态，做出每个选择需要对应代价的，刚才说过这个代价是因人而异的，所以我需要给自己定制一个损失函数，计算不同指法切换的「别扭程度」。

现在的问题就变成了一个标准的动态规划问题，根据损失函数做出「别扭程度」最小的选择，使得整段演奏最流畅……

当然，最后这个算法时间复杂度太高了，我们刚才分析的只是单个的音符，但如果串成曲子，时空复杂度还得再乘曲子的音符数，很大。

而且，这个损失函数很难量化，钢琴的黑白键命中难度不同，而且「别扭程度」只能靠感觉，有点不严谨……

不过，本就没必要计算整首曲子的指法，只需要计算某些复杂段落的指法即可，这个算法还是比较有效的。

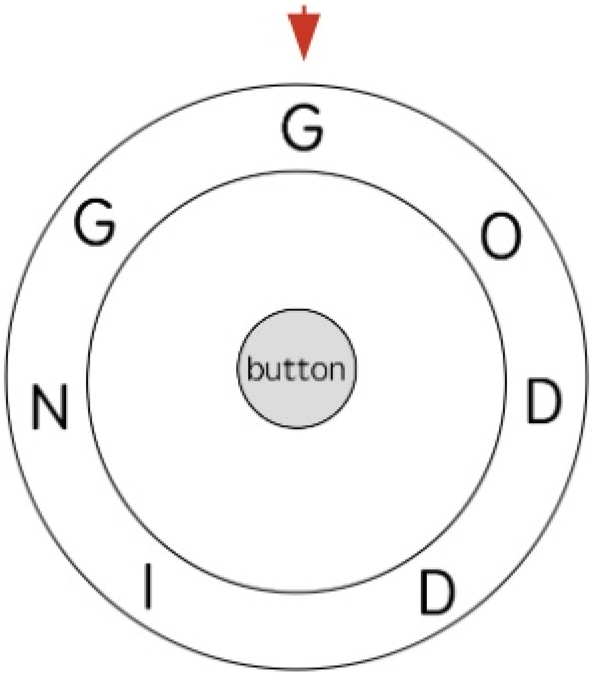
扯了这么多题外话终于要步入正题了，今天要讲的力扣第 514 题「自由之路」和钢琴指法问题有异曲同工之妙，如果你能理解钢琴的例子，相信你也能很快做出这道算法题。

题目给你输入一个字符串 `ring` 代表圆盘上的字符（指针位置在 12 点钟方向，初始指向 `ring[0]`），再输入一个字符串 `key` 代表你需要拨动圆盘输入的字符串，你的算法需要返回输入这个 `key` 至少进行多少次操作（拨动一格圆盘和按下圆盘中间的按钮都算是一次操作）。

函数签名如下：

```
int findRotateSteps(String ring, String key);
```

比如题目举的例子，输入 `ring = "godding", key = "gd"`，对应的圆盘如下（大写只是为了清晰，实际上输入的字符串都是小写字母）：



我们需要输入 `key = "gd"`，算法返回 4。

因为现在指针指的字母就是字母 `"g"`，所以可以直接按下中间的按钮，然后再将圆盘逆时针拨动两格，让指针指向字母 `"d"`，然后再按一次中间的按钮。

上述过程，按了两次按钮，拨了两格转盘，总共操作了 4 次，是最少的操作次数，所以算法应该返回 4。

我们这里可以首先给题目做一个等价，转动圆盘是不是就等于拨动指针？

原题可以转化为：圆盘固定，我们可以拨动指针；现在需要我们拨动指针并按下按钮，以最少的操作次数输入 `key` 对应的字符串。

那么，这个问题如何使用动态规划的技巧解决呢？或者说，这道题的「状态」和「选择」是什么呢？

「状态」就是「当前需要输入的字符」和「当前圆盘指针的位置」。

再具体点，「状态」就是 `i` 和 `j` 两个变量。我们可以用 `i` 表示当前圆盘上指针指向的字符（也就是 `ring[i]`）；用 `j` 表示需要输入的字符（也就是 `key[j]`）。

这样我们可以写这样一个 `dp` 函数：

```
int dp(String ring, int i, String key, int j);
```

这个 `dp` 函数的定义如下：

当圆盘指针指向 `ring[i]` 时，输入字符串 `key[j..]` 至少需要 `dp(ring, i, key, j)` 次操作。

根据这个定义，题目其实就是想计算 `dp(ring, 0, key, 0)` 的值，而且我们可以把 `dp` 函数的 base case 写出来：

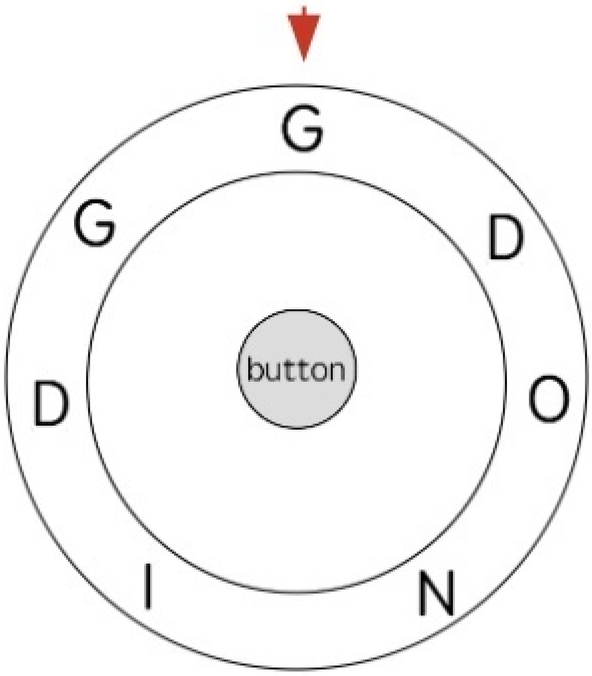
```
int dp(String ring, int i, String, int j) {
    // base case, 完成输入
    if (j == key.length()) return 0;
    // ...
}
```

接下来，思考一下如何根据状态做选择，如何进行状态转移？

「选择」就是「如何拨动指针得到待输入的字符」。

再具体点就是，对于现在想输入的字符 `key[j]`，我们可以如何拨动圆盘，得到这个字符？

比如说输入 `ring = "gdonidg"`，现在圆盘的状态如下图：



假设我想输入的字符 `key[j] = "d"`，圆盘中有两个字母 `"d"`，而且我可以顺时针也可以逆时针拨动指针，所以总共有四种「选择」输入字符 `"d"`，我们需要选择操作次数最少的那个拨法。

大致的代码逻辑如下：

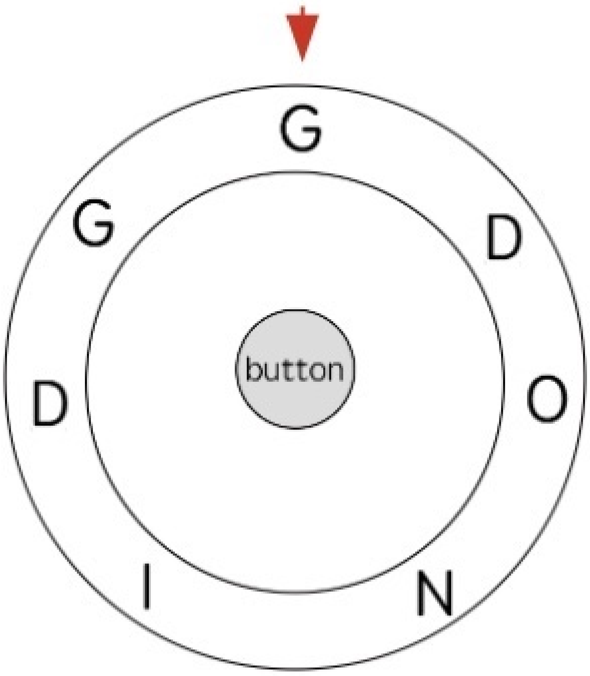
```
int dp(String ring, int i, String, int j) {
    // base case 完成输入
    if (j == key.length()) return 0;

    // 做选择
    int res = Integer.MAX_VALUE;
    for (int k : [字符 key[j] 在 ring 中的所有索引]) {
        res = min(
            把 i 顺时针转到 k 的代价,
            把 i 逆时针转到 k 的代价
        );
    }
```

```
    return res;
}
```

至于到底是顺时针还是逆时针，其实非常好判断，怎么近就怎么来；但是对于圆盘中的两个字符 "d"，还能是怎么近怎么来吗？

不能，因为这和 key[i] 之后需要输入的字符有关，还是上面的例子：



如果输入的是 key = "di"，那么即便右边的 "d" 离得近，也应该去左边的 "d"，因为左边的 "d" 旁边就是 "i"，「整体」的操作数最少。

那么，应该如何判断呢？其实就是穷举，递归调用 dp 函数，把两种选择的「整体」代价算出来，然后再做比较就行了。

讲到这就差不多了，直接看代码吧：

```
// 字符 -> 索引列表
HashMap<Character, List<Integer>> charToIndex = new HashMap<>();
// 备忘录
int[][] memo;

/* 主函数 */
int findRotateSteps(String ring, String key) {
    int m = ring.length();
    int n = key.length();
    // 备忘录全部初始化为 0
    memo = new int[m][n];
    // 记录圆环上字符到索引的映射
    for (int i = 0; i < ring.length(); i++) {
        char c = ring.charAt(i);
```

```
            if (!charToIndex.containsKey(c)) {
                charToIndex.put(c, new LinkedList<>());
            }
            charToIndex.get(c).add(i);
        }
        // 圆盘指针最初指向 12 点钟方向，
        // 从第一个字符开始输入 key
        return dp(ring, 0, key, 0);
    }

    // 计算圆盘指针在 ring[i]，输入 key[j..] 的最少操作数
    int dp(String ring, int i, String key, int j) {
        // base case 完成输入
        if (j == key.length()) return 0;
        // 查找备忘录，避免重叠子问题
        if (memo[i][j] != 0) return memo[i][j];

        int n = ring.length();
        // 做选择
        int res = Integer.MAX_VALUE;
        // ring 上可能有多个字符 key[j]
        for (int k : charToIndex.get(key.charAt(j))) {
            // 拨动指针的次数
            int delta = Math.abs(k - i);
            // 选择顺时针还是逆时针
            delta = Math.min(delta, n - delta);
            // 将指针拨到 ring[k]，继续输入 key[j+1..]
            int subProblem = dp(ring, k, key, j + 1);
            // 选择「整体」操作次数最少的
            // 加一是因为按动按钮也是一次操作
            res = Math.min(res, 1 + delta + subProblem);
        }
        // 将结果存入备忘录
        memo[i][j] = res;
        return res;
    }
```

这段代码是 C++ 写的，因为我觉得涉及字符串的算法 C++ 更方便一些，这里说一些语言相关的细节问题：

1、`unordered_map` 就是哈希表，当访问不存在的键时，会自动创建对应的值，所以可以直接 `push_back` 而不用担心空指针错误。

2、`min` 函数的参数都是 int 型，所以必须先用一个 int 型变量 `n` 存储 `ring.size()`，然后调用 `min(delta, n - delta)`，否则会报错。

至此，这道题就解决了。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

旅游省钱大法：加权最短路径

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

787. K 站中转内最便宜的航班（中等）

毕业季，对过去也许有些欢乐和感伤，对未来也许有些迷茫和向往，不过这些终究是过眼云烟，迟早会被时间淡化和遗忘。

在这段美好时光的末尾，确实应该来一场说走就走的毕业旅行，放肆一把，给青春画上一个完美的句号。

那么，本文就教给你一个动态规划算法，在~~毕业旅行中省钱~~ 节约追求诗和远方的资本。



假设，你准备从学校所在的城市出发，游历多个城市，一路浪到公司入职，那么你应该如何安排旅游路线，才能最小化机票的开销？

我们来看看力扣第 787 题「K 站中转内最便宜的航班」，我描述一下题目：

现在有 `n` 个城市，分别用 `0`, `1`..., `n - 1` 这些序号表示，城市之间的航线用三元组 `[from, to, price]` 来表示，比如说三元组 `[0,1,100]` 就表示，从城市 `0` 到城市 `1` 之间的机票价格是 100 元。

题目会给你输入若干参数：正整数 n 代表城市个数，数组 flights 装着若干三元组代表城市间的航线及价格，城市编号 src 代表你所在的城市，城市编号 dst 代表你要去的目标城市，整数 K 代表你最多经过的中转站个数。

函数签名如下：

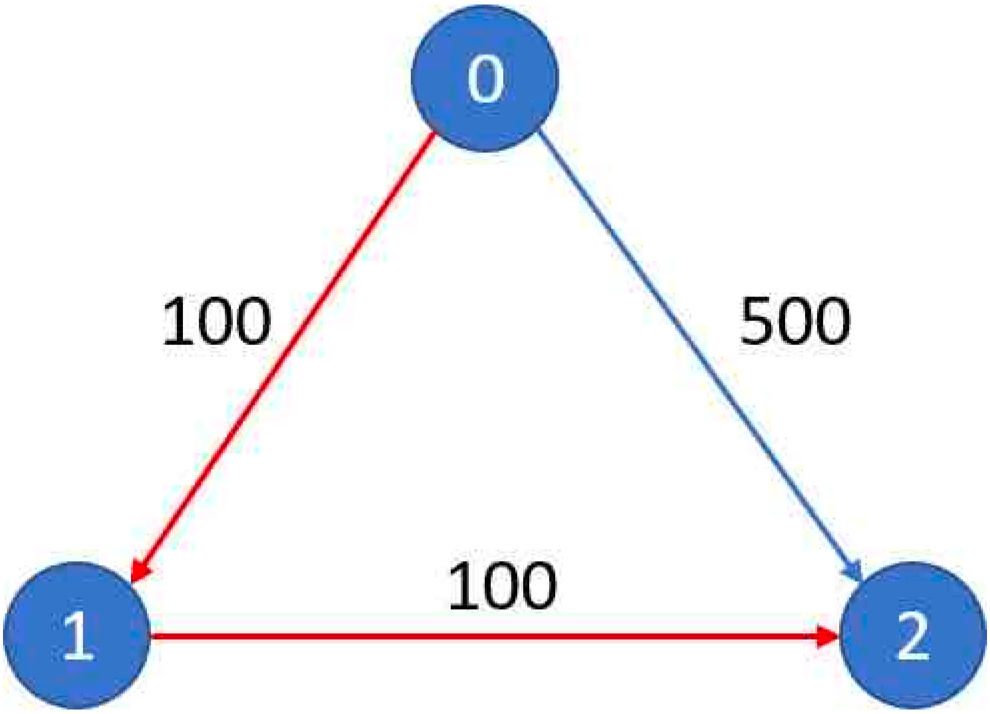
```
int findCheapestPrice(int n, int[][] flights, int src, int dst, int K);
```

请你的算法计算，在 K 次中转之内，从 src 到 dst 所需的最小花费是多少钱，如果无法到达，则返回 -1。

比方说题目给的例子：

`n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, K = 1`

航线就是如下这张图所示，有向边代表航向的方向，边上的数字代表航线的机票价格：



出发点是 0，到达点是 2，允许的最大中转次数 K 为 1，所以最小的开销就是图中红色的两条边，从 0 出发，经过中转城市 1 到达目标城市 2，所以算法的返回值应该是 200。

注意这个中转次数的上限 K 是比较棘手的，如果上述题目将 K 改为 0，也就是不允许中转，那么我们的算法只能返回 500 了，也就是直接从 0 飞到 2。

很明显，这题就是个加权有向图中求最短路径的问题。

说白了，就是给你一幅加权有向图，让你求 src 到 dst 权重最小的一条路径，同时要满足，**这条路径最多不能超过 K + 1 条边**（经过 K 个节点相当于经过 K + 1 条边）。

我们来分析下求最短路径相关的算法。

BFS 算法思路

我们前文 BFS 算法框架详解 中说到，求最短路径，肯定可以用 BFS 算法来解决。

因为 BFS 算法相当于从起始点开始，一步一步向外扩散，那当然是离起点越近的节点越先被遍历到，如果 BFS 遍历的过程中遇到终点，那么走的肯定是最短路径。

不过呢，我们在 BFS 算法框架详解 用的是普通的队列 `Queue` 来遍历多叉树，而对于加权图的最短路径来说，普通的队列不管用了，得用优先级队列 `PriorityQueue`。

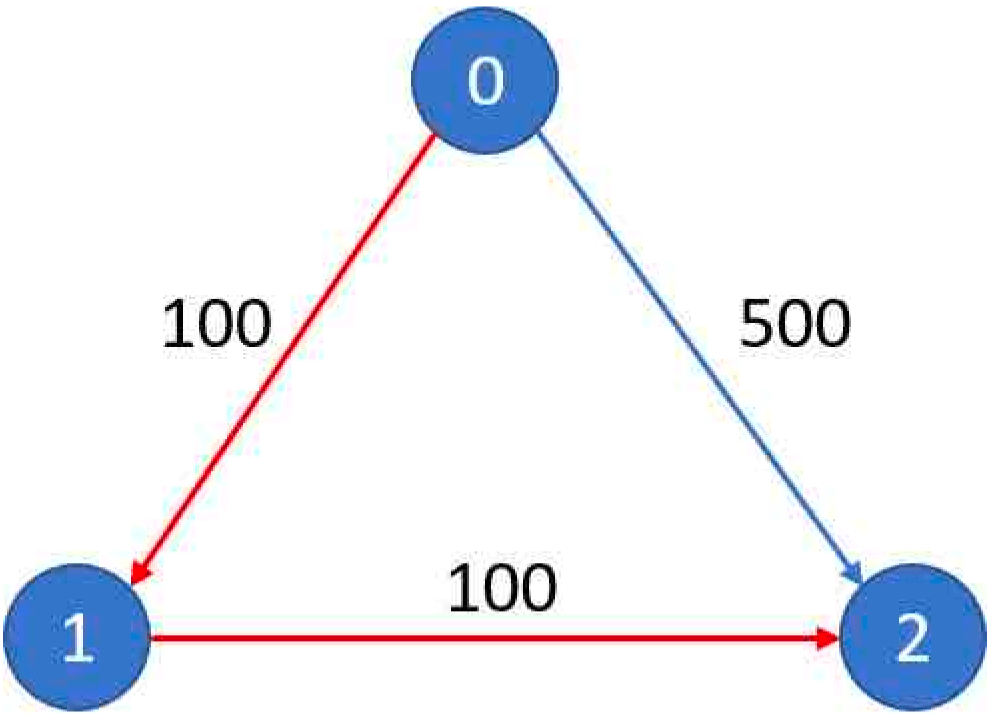
为什么呢？也好理解，在多叉树（或者扩展到无权图）的遍历中，与其说边没有权重，不如说每条边的权重都是 1，起点与终点之间的路径权重就是它们之间「边」的条数。

这样，按照 BFS 算法一步步向四周扩散的逻辑，先遍历到的节点和起点之间的「边」更少，累计的权重当然少。

换言之，先进入 `Queue` 的节点就是离起点近的，路径权重小的节点。

但对于加权图，路径中边的条数和路径的权重并不是正相关的关系了，有的路径可能边的条数很少，但每条边的权重都很大，那显然这条路径权重也会很大，很难成为最短路径。

比如题目给的这个例子：



你是可以一步从 `0` 走到 `2`，但路径权重不见得是最小的。

所以，对于加权图的场景，我们需要优先级队列「自动排序」的特性，将路径权重较小的节点排在队列前面，以此为基础施展 BFS 算法，也就变成了 Dijkstra 算法。

说了这么多 BFS 算法思路，只是帮助大家融会贯通一下，我们本文准备用动态规划来解决这道题，因为我们公众号好久没有写动态规划相关的算法了，关于 Dijkstra 算法的实现代码，文末有写，供读者参考。

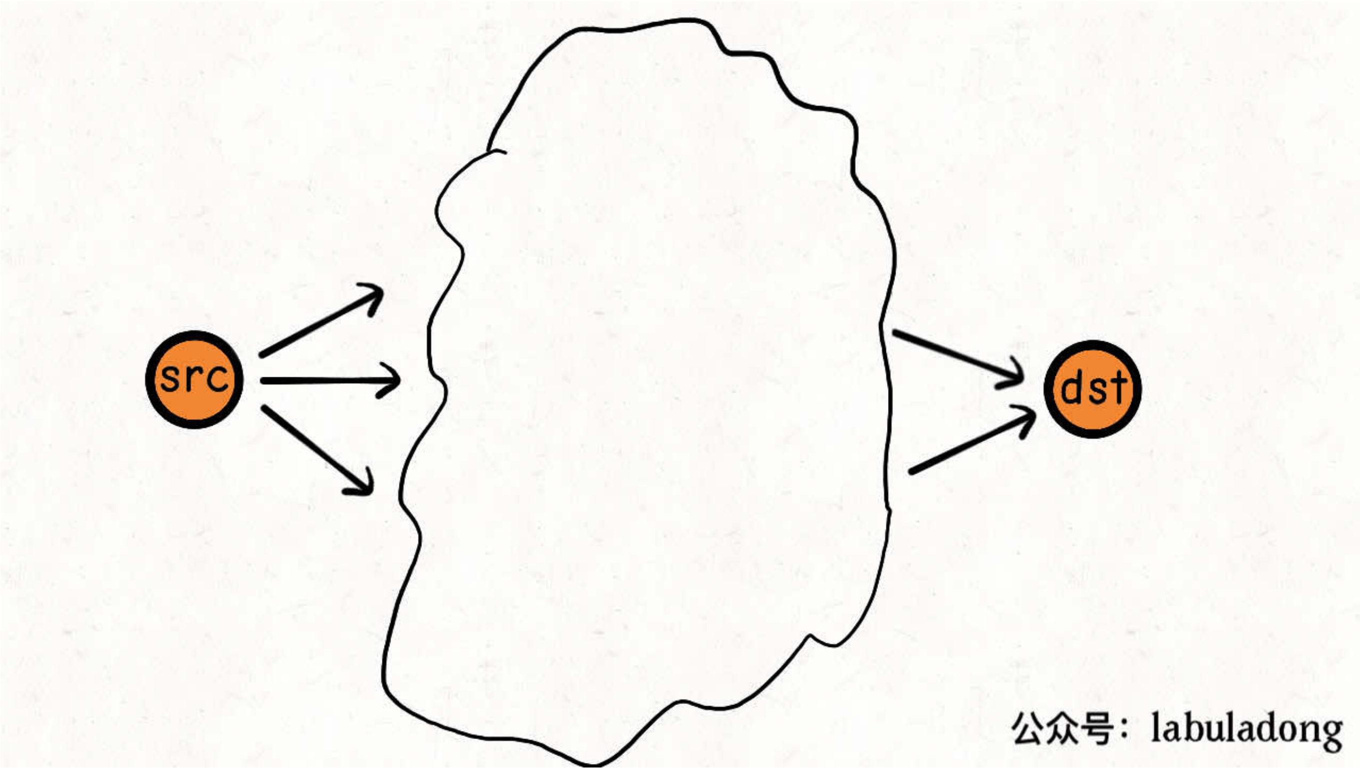
动态规划思路

我们前文 动态规划核心套路详解 中说过，求最值的问题，很多都可能使用动态规划来求解。

加权最短路径问题，再加个 K 的限制也无妨，不也是个求最值的问题嘛，动态规划统统拿下。

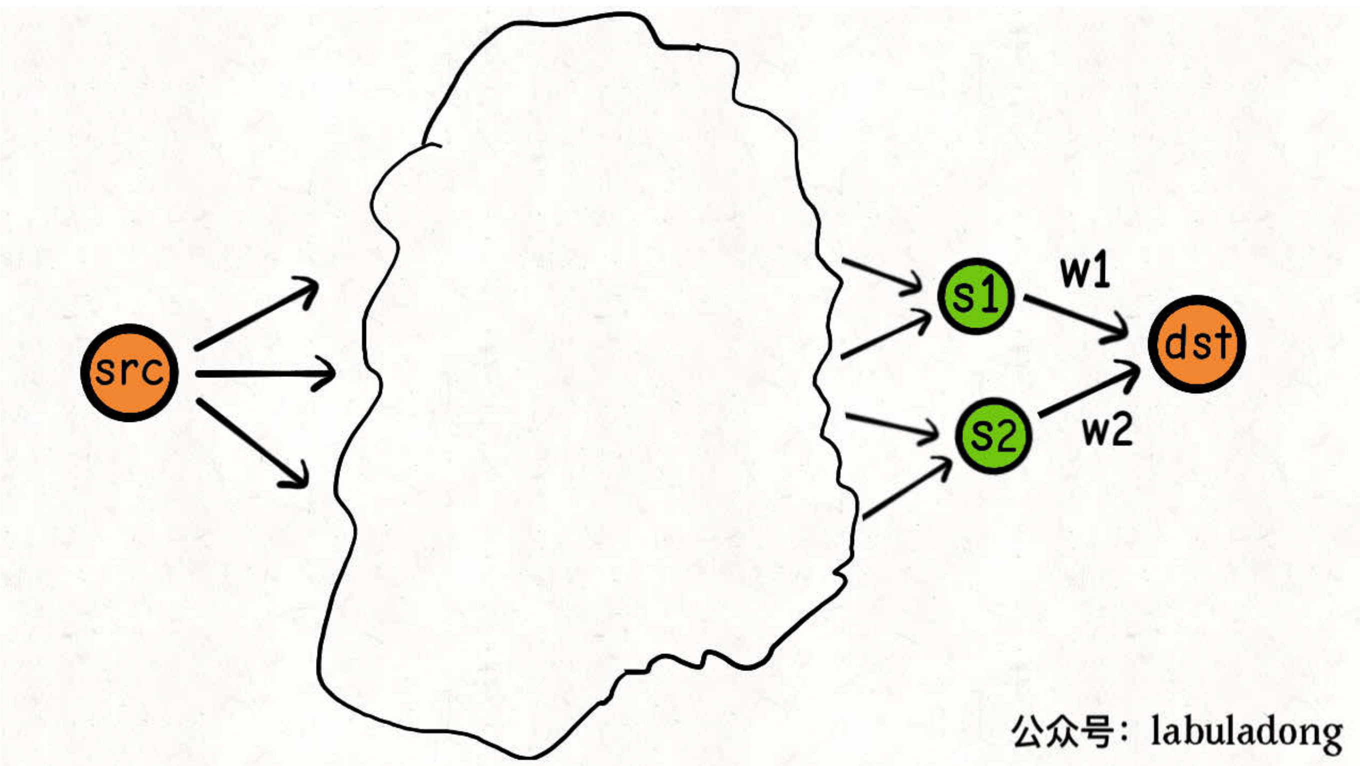
我们先不管 K 的限制，但就「加权最短路径」这个问题来看看，它怎么就是个动态规划问题了呢？

比方说，现在我想计算 src 到 dst 的最短路径：



最小权重是多少？我不知道。

但我可以把问题进行分解：



s1, s2 是指向 dst 的相邻节点，它们之间的权重我是知道的，分别是 w1, w2。

只要我知道了从 src 到 s1, s2 的最短路径，我不就知道 src 到 dst 的最短路径了吗！

```
minPath(src, dst) = min(
    minPath(src, s1) + w1,
    minPath(src, s2) + w2
)
```

这其实就是递归关系了，就是这么简单。

不过别忘了，题目对我们的最短路径还有个「路径上不能超过 K + 1 条边」的限制。

那么我们不妨定义这样一个 dp 函数：

```
int dp(int s, int k);
```

函数的定义如下：

从起点 src 出发，k 步之内（一步就是一条边）到达节点 s 的最小路径权重为 dp(s, k)。

那么，dp 函数的 base case 就显而易见了：

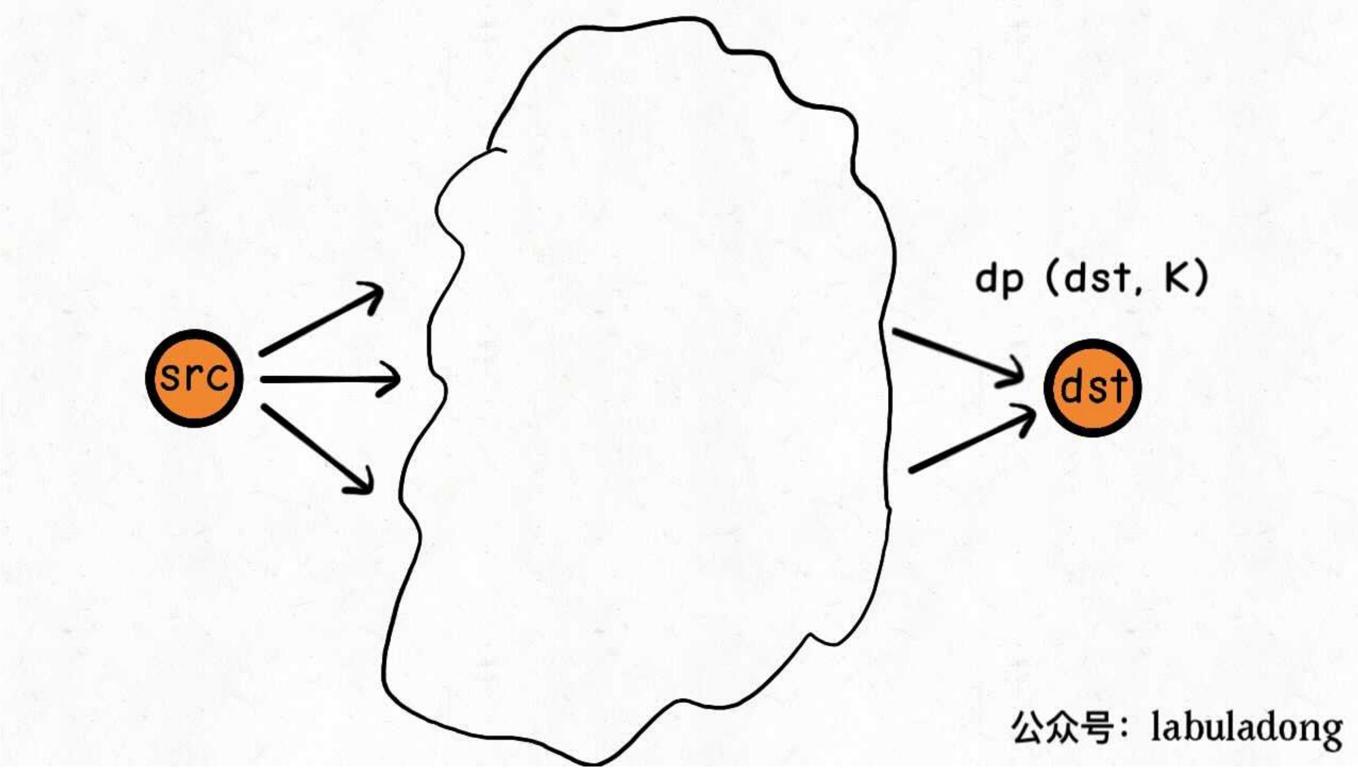
```
// 定义：从 src 出发，k 步之内到达 s 的最小成本
    int dp(int s, int k) {
        // 从 src 到 src，一步都不用走
        if (s == src) {
            return 0;
        }
        // 如果步数用尽，就无解了
        if (k == 0) {
            return -1;
        }

        // ...
    }
```

题目想求的最小机票开销就可以用 dp(dst, K+1) 来表示：

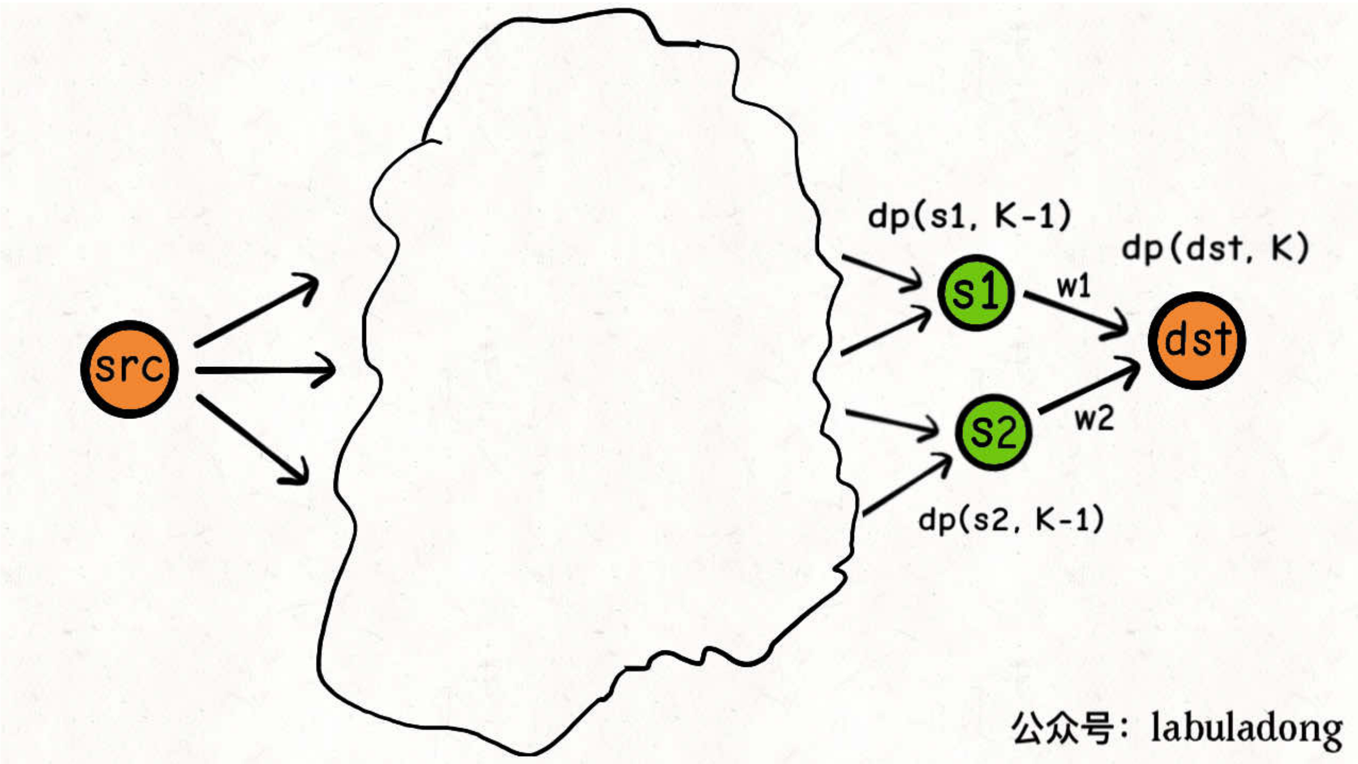
```
int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {
    // 将中转站个数转化成边的条数
    K++;
    //...
    return dp(dst, K);
```

添加了一个 K 条边的限制，状态转移方程怎么写呢？其实和刚才是一样的：



K 步之内从 src 到 dst 的最小路径权重是多少？我不知道。

但我可以把问题分解：



s1, s2 是指向 dst 的相邻节点，我只要知道 K - 1 步之内从 src 到达 s1, s2，那我就可以在 K 步之内从 src 到达 dst。

也就是如下关系式：

```
dp(dst, k) = min(
    dp(s1, k - 1) + w1,
    dp(s2, k - 1) + w2
)
```

这就是新的状态转移方程，如果你能看懂这个算式，就已经可以解决这道题了。

代码实现

根据上述思路，我怎么知道 `s1, s2` 是指向 `dst` 的相邻节点，他们之间的权重是 `w1, w2`?

我希望给一个节点，就能知道有谁指向这个节点，还知道它们之间的权重，对吧。

专业点说，得用一个数据结构记录每个节点的「入度」indegree：

```
// 哈希表记录每个点的入度
// to -> [from, price]
HashMap<Integer, List<int[]>> indegree;
int src, dst;

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int
K) {
    // 将中转站个数转化成边的条数
    K++;
    this.src = src;
    this.dst = dst;

    indegree = new HashMap<>();
    for (int[] f : flights) {
        int from = f[0];
        int to = f[1];
        int price = f[2];
        // 记录谁指向该节点，以及之间的权重
        indegree.putIfAbsent(to, new LinkedList<>());
        indegree.get(to).add(new int[] {from, price});
    }

    return dp(dst, K);
}
```

有了 `indegree` 存储入度，那么就可以具体实现 `dp` 函数了：

```
// 定义：从 src 出发，k 步之内到达 s 的最短路径权重
int dp(int s, int k) {
    // base case
    if (s == src) {
        return 0;
    }
    if (k == 0) {
```

```java
            return -1;
        }
        // 初始化为最大值，方便等会取最小值
        int res = Integer.MAX_VALUE;
        if (indegree.containsKey(s)) {
            // 当 s 有入度节点时，分解为子问题
            for (int[] v : indegree.get(s)) {
                int from = v[0];
                int price = v[1];
                // 从 src 到达相邻的入度节点所需的最短路径权重
                int subProblem = dp(from, k - 1);
                // 跳过无解的情况
                if (subProblem != -1) {
                    res = Math.min(res, subProblem + price);
                }
            }
        }
        // 如果还是初始值，说明此节点不可达
        return res == Integer.MAX_VALUE ? -1 : res;
    }
```

有之前的铺垫，这段解法逻辑应该是很清晰的。当然，对于动态规划问题，肯定要消除重叠子问题。

为什么有重叠子问题？很简单，如果某个节点同时指向两个其他节点，那么这两个节点就有相同的一个入度节点，就会产生重复的递归计算。

怎么消除重叠子问题？找问题的「状态」。

状态是什么？在问题分解（状态转移）的过程中变化的，就是状态。

谁在变化？显然就是 `dp` 函数的参数 `s` 和 `k`，每次递归调用，目标点 `s` 和步数约束 `k` 在变化。

所以，本题的状态有两个，应该算是二维动态规划，我们可以用一个 `memo` 二维数组或者哈希表作为备忘录，减少重复计算。

我们选用二维数组做备忘录吧，注意 `K` 是从 1 开始算的，所以备忘录初始大小要再加一：

```java
int src, dst;
HashMap<Integer, List<int[]>> indegree;
// 备忘录
int[][] memo;

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int
K) {
    K++;
    this.src = src;
    this.dst = dst;
    // 初始化备忘录，全部填一个特殊值
    memo = new int[n][K + 1];
    for (int[] row : memo) {
        Arrays.fill(row, -888);
    }
```

```
    // 其他不变
    // ...

    return dp(dst, K);
}

// 定义：从 src 出发，k 步之内到达 s 的最小成本
int dp(int s, int k) {
    // base case
    if (s == src) {
        return 0;
    }
    if (k == 0) {
        return -1;
    }
    // 查备忘录，防止冗余计算
    if (memo[s][k] != -888) {
        return memo[s][k];
    }

    // 状态转移不变
    // ...

    // 存入备忘录
    memo[s][k] = res == Integer.MAX_VALUE ? -1 : res;
    return memo[s][k];
}
```

备忘录初始值为啥初始为 -888？前文 base case 和备忘录的初始值怎么定 说过，随便初始化一个无意义的值就行。

至此，这道题就通过自顶向下的递归方式解决了。当然，完全可以按照这个解法衍生出自底向上迭代的动态规划解法，但由于篇幅所限，我就不写了，反正本质上都是一样的。

其实，大家如果把我们号之前的所有动态规划文章都看一遍，就会发现我们一直在套用 动态规划核心套路，其实真没什么困难的。

最后扩展一下，有的读者可能会问：既然这个问题本质上是一个图的遍历问题，为什么不需要 `visited` 集合记录已经访问过的节点？

这个问题我在 Dijkstra 算法模板 中探讨过，可以去看看。另外，这题也可以利用 Dijkstra 算法模板来解决，代码如下：

```
public int findCheapestPrice(int n, int[][] flights, int src, int dst, int
K) {
    List<int[]>[] graph = new LinkedList[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new LinkedList<>();
    }
    for (int[] edge : flights) {
```

```java
            int from = edge[0];
            int to = edge[1];
            int price = edge[2];
            graph[from].add(new int[]{to, price});
        }

        // 启动 dijkstra 算法
        // 计算以 src 为起点在 k 次中转到达 dst 的最短路径
        K++;
        return dijkstra(graph, src, K, dst);
    }

    class State {
        // 图节点的 id
        int id;
        // 从 src 节点到当前节点的花费
        int costFromSrc;
        // 从 src 节点到当前节点经过的节点个数
        int nodeNumFromSrc;

        State(int id, int costFromSrc, int nodeNumFromSrc) {
            this.id = id;
            this.costFromSrc = costFromSrc;
            this.nodeNumFromSrc = nodeNumFromSrc;
        }
    }

    // 输入一个起点 src，计算从 src 到其他节点的最短距离
    int dijkstra(List<int[]>[] graph, int src, int k, int dst) {
        // 定义：从起点 src 到达节点 i 的最短路径权重为 distTo[i]
        int[] distTo = new int[graph.length];
        // 定义：从起点 src 到达节点 i 至少要经过 nodeNumTo[i] 个节点
        int[] nodeNumTo = new int[graph.length];
        Arrays.fill(distTo, Integer.MAX_VALUE);
        Arrays.fill(nodeNumTo, Integer.MAX_VALUE);
        // base case
        distTo[src] = 0;
        nodeNumTo[src] = 0;

        // 优先级队列，costFromSrc 较小的排在前面
        Queue<State> pq = new PriorityQueue<>((a, b) -> {
            return a.costFromSrc - b.costFromSrc;
        });
        // 从起点 src 开始进行 BFS
        pq.offer(new State(src, 0, 0));

        while (!pq.isEmpty()) {
            State curState = pq.poll();
            int curNodeID = curState.id;
            int costFromSrc = curState.costFromSrc;
            int curNodeNumFromSrc = curState.nodeNumFromSrc;

            if (curNodeID == dst) {
                // 找到最短路径
```

```
                return costFromSrc;
            }
            if (curNodeNumFromSrc == k) {
                // 中转次数耗尽
                continue;
            }

            // 将 curNode 的相邻节点装入队列
            for (int[] neighbor : graph[curNodeID]) {
                int nextNodeID = neighbor[0];
                int costToNextNode = costFromSrc + neighbor[1];
                // 中转次数消耗 1
                int nextNodeNumFromSrc = curNodeNumFromSrc + 1;

                // 更新 dp table
                if (distTo[nextNodeID] > costToNextNode) {
                    distTo[nextNodeID] = costToNextNode;
                    nodeNumTo[nextNodeID] = nextNodeNumFromSrc;
                }
                // 剪枝，如果中转次数更多，花费还更大，那必然不会是最短路径
                if (costToNextNode > distTo[nextNodeID]
                    && nextNodeNumFromSrc > nodeNumTo[nextNodeID]) {
                    continue;
                }

                pq.offer(new State(nextNodeID, costToNextNode,
nextNodeNumFromSrc));
            }
        }
        return -1;
}
```

关于这个解法这里就不多解释了，可对照前文 Dijkstra 算法模板 理解。

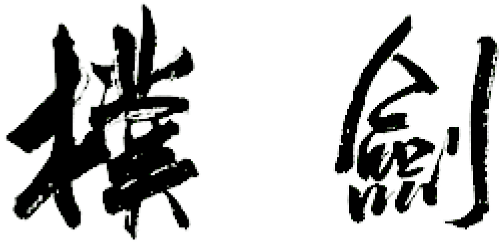
如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



朴剑篇、其他经典算法



除了像动态规划、回溯算法这种容易归类的算法类型，还有很多其他算法题并没有特别明显的特征，或者很难将一系列题目抽象汇总到一个算法技巧之下。

对于这类问题，只能说多做多总结，增加对这些算法问题的积累。

5.2 数学算法

数学算法是一个很大的范畴，要说高深的话可以非常高深，不过从刷题的角度来说，数学算法大多就是位运算、找规律、概率算法、素数之类的考点。

公众号标签：数学算法

如何高效寻找素数

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

204. 计数质数（简单）

素数的定义看起来很简单，如果一个数如果只能被 1 和它本身整除，那么这个数就是素数。

虽然素数的定义并不复杂，恐怕没多少人真的能把素数相关的算法写得高效。

比如力扣第 204 题「计数质数」，让你写这样一个函数：

```
// 返回区间 [2, n) 中有几个素数
int countPrimes(int n)

// 比如 countPrimes(10) 返回 4
// 因为 2,3,5,7 是素数
```

你会如何写这个函数？我想大家应该会这样写：

```
int countPrimes(int n) {
    int count = 0;
    for (int i = 2; i < n; i++)
        if (isPrime(i)) count++;
    return count;
}

// 判断整数 n 是否是素数
boolean isPrime(int n) {
    for (int i = 2; i < n; i++)
        if (n % i == 0)
            // 有其他整除因子
            return false;
    return true;
}
```

这样写的话时间复杂度 O(n^2)，问题很大。**首先你用 isPrime 函数来辅助的思路就不够高效；而且就算你要用 isPrime 函数，这样写算法也是存在计算冗余的。**

先来简单说下**如果你要判断一个数是不是素数，应该如何写算法**。只需稍微修改一下上面的 isPrime 代码中的 for 循环条件：

```
boolean isPrime(int n) {
    for (int i = 2; i * i <= n; i++)
        ...
}
```

换句话说，`i` 不需要遍历到 `n`，而只需要到 `sqrt(n)` 即可。为什么呢，我们举个例子，假设 `n = 12`。

```
12 = 2 × 6
12 = 3 × 4
12 = sqrt(12) × sqrt(12)
12 = 4 × 3
12 = 6 × 2
```

可以看到，后两个乘积就是前面两个反过来，反转临界点就在 `sqrt(n)`。

换句话说，如果在 `[2,sqrt(n)]` 这个区间之内没有发现可整除因子，就可以直接断定 `n` 是素数了，因为在区间 `[sqrt(n),n]` 也一定不会发现可整除因子。

现在，`isPrime` 函数的时间复杂度降为 O(sqrt(N))，**但是我们实现 `countPrimes` 函数其实并不需要这个函数**，以上只是希望读者明白 `sqrt(n)` 的含义，因为等会还会用到。

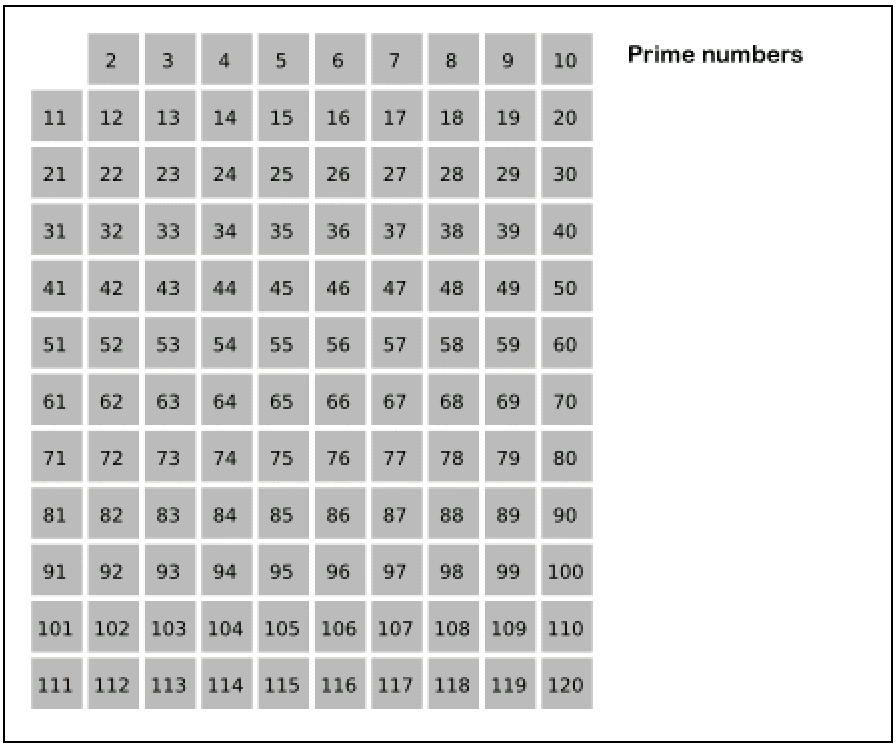
高效实现 `countPrimes`

高效解决这个问题的核心思路是和上面的常规思路反着来：

首先从 2 开始，我们知道 2 是一个素数，那么 2 × 2 = 4, 3 × 2 = 6, 4 × 2 = 8... 都不可能是素数了。

然后我们发现 3 也是素数，那么 3 × 2 = 6, 3 × 3 = 9, 3 × 4 = 12... 也都不可能是素数了。

这个思路又叫做「筛选法」，Wikipedia 的这个 GIF 很形象：



看到这里，你是否有点明白这个排除法的逻辑了呢？先看我们的第一版代码：

```
int countPrimes(int n) {
    boolean[] isPrime = new boolean[n];
    // 将数组都初始化为 true
    Arrays.fill(isPrime, true);

    for (int i = 2; i < n; i++)
        if (isPrime[i])
            // i 的倍数不可能是素数了
            for (int j = 2 * i; j < n; j += i)
                    isPrime[j] = false;

    int count = 0;
    for (int i = 2; i < n; i++)
        if (isPrime[i]) count++;

    return count;
}
```

如果上面这段代码你能够理解，那么你已经掌握了整体思路，但是还有两个细微的地方可以优化。

首先，回想刚才判断一个数是否是素数的 `isPrime` 函数，由于因子的对称性，其中的 for 循环只需要遍历 `[2,sqrt(n)]` 就够了。这里也是类似的，我们外层的 for 循环也只需要遍历到 `sqrt(n)`：

```
for (int i = 2; i * i < n; i++)
    if (isPrime[i])
        ...
```

除此之外，很难注意到内层的 for 循环也可以优化。我们之前的做法是：

```
for (int j = 2 * i; j < n; j += i)
    isPrime[j] = false;
```

这样可以把 `i` 的整数倍都标记为 `false`，但是仍然存在计算冗余。

比如 `n = 25`，`i = 4` 时算法会标记 4 × 2 = 8，4 × 3 = 12 等等数字，但是这两个数字已经被 `i = 2` 和 `i = 3` 的 2 × 4 和 3 × 4 标记了。

我们可以稍微优化一下，让 `j` 从 `i` 的平方开始遍历，而不是从 `2 * i` 开始：

```
for (int j = i * i; j < n; j += i)
    isPrime[j] = false;
```

这样，素数计数的算法就高效实现了，其实这个算法有一个名字，叫做 Sieve of Eratosthenes。看下完整的最终代码：

```
int countPrimes(int n) {
    boolean[] isPrime = new boolean[n];
    Arrays.fill(isPrime, true);
    for (int i = 2; i * i < n; i++)
        if (isPrime[i])
            for (int j = i * i; j < n; j += i)
                isPrime[j] = false;

    int count = 0;
    for (int i = 2; i < n; i++)
        if (isPrime[i]) count++;

    return count;
}
```

该算法的时间复杂度比较难算，显然时间跟这两个嵌套的 for 循环有关，其操作数应该是：

$n/2 + n/3 + n/5 + n/7 + ... = n \times (1/2 + 1/3 + 1/5 + 1/7...)$

括号中是素数的倒数。其最终结果是 O(N * loglogN)，有兴趣的读者可以查一下该算法的时间复杂度证明。

以上就是素数算法相关的全部内容。怎么样，是不是看似简单的问题却有不少细节可以打磨呀？

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

两道常考的阶乘算法题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

172. 阶乘后的零（简单）

793. 阶乘后 K 个零（困难）

笔试题中经常看到阶乘相关的题目，今天说两个最常见的题目：

1、输入一个非负整数 `n`，请你计算阶乘 `n!` 的结果末尾有几个 0。

这也是力扣第 172 题「阶乘后的零」，比如说输入 `n = 5`，算法返回 1，因为 `5! = 120`，末尾有一个 0。

函数签名如下：

```
int trailingZeroes(int n);
```

2、输入一个非负整数 `K`，请你计算有多少个 `n`，满足 `n!` 的结果末尾恰好有 `K` 个 0。

这也是力扣第 793 题「阶乘后 K 个零」，比如说输入 `K = 1`，算法返回 5，因为 `5!,6!,7!,8!,9!` 这 5 个阶乘的结果最后只有一个 0，即有 5 个 `n` 满足条件。

函数签名如下：

```
int preimageSizeFZF(int K);
```

我把这两个题放在一起，肯定是因为它们有共性，下面我们来逐一分析。

题目一

肯定不可能真去把 `n!` 的结果算出来，阶乘增长可是比指数增长都恐怖，趁早死了这条心吧。

那么，结果的末尾的 0 从哪里来的？我们有没有投机取巧的方法计算出来？

首先，两个数相乘结果末尾有 0，一定是因为两个数中有因子 2 和 5，因为 10 = 2 x 5。

也就是说，问题转化为：`n!` 最多可以分解出多少个因子 2 和 5？

比如说 `n = 25`，那么 `25!` 最多可以分解出几个 2 和 5 相乘？这个主要取决于能分解出几个因子 5，因为每个偶数都能分解出因子 2，因子 2 肯定比因子 5 多得多。

`25!` 中 5 可以提供一个，10 可以提供一个，15 可以提供一个，20 可以提供一个，25 可以提供两个，总共有 6 个因子 5，所以 `25!` 的结果末尾就有 6 个 0。

现在，问题转化为：`n!` 最多可以分解出多少个因子 5？

难点在于像 25，50，125 这样的数，可以提供不止一个因子 5，怎么才能不漏掉呢？

这样，我们假设 `n = 125`，来算一算 `125!` 的结果末尾有几个 0：

首先，125 / 5 = 25，这一步就是计算有多少个像 5，15，20，25 这些 5 的倍数，它们一定可以提供一个因子 5。

但是，这些足够吗？刚才说了，像 25，50，75 这些 25 的倍数，可以提供两个因子 5，那么我们再计算出 `125!` 中有 125 / 25 = 5 个 25 的倍数，它们每人可以额外再提供一个因子 5。

够了吗？我们发现 125 = 5 x 5 x 5，像 125，250 这些 125 的倍数，可以提供 3 个因子 5，那么我们还得再计算出 `125!` 中有 125 / 125 = 1 个 125 的倍数，它还可以额外再提供一个因子 5。

这下应该够了，`125!` 最多可以分解出 25 + 5 + 1 = 31 个因子 5，也就是说阶乘结果的末尾有 31 个 0。

理解了这个思路，就可以理解解法代码了：

```
int trailingZeroes(int n) {
    int res = 0;
    long divisor = 5;
    while (divisor <= n) {
        res += n / divisor;
        divisor *= 5;
    }
    return res;
}
```

这里 `divisor` 变量使用 long 型，因为假如 `n` 比较大，考虑 while 循环的结束条件，`divisor` 可能出现整型溢出。

上述代码可以改写地更简单一些：

```
int trailingZeroes(int n) {
    int res = 0;
    for (int d = n; d / 5 > 0; d = d / 5) {
        res += d / 5;
    }
    return res;
}
```

这样，这道题就解决了，时间复杂度是底数为 5 的对数，也就是 `O(logN)`，我们看看下如何基于这道题的解法完成下一道题目。

第二题

现在是给你一个非负整数 `K`，问你有多少个 `n`，使得 `n!` 结果末尾有 `K` 个 0。

一个直观地暴力解法就是穷举呗，因为随着 `n` 的增加，`n!` 肯定是递增的，`trailingZeroes(n!)` 肯定也是递增的，伪码逻辑如下：

```
int res = 0;
for (int n = 0; n < +inf; n++) {
    if (trailingZeroes(n) < K) {
        continue;
    }
    if (trailingZeroes(n) > K) {
        break;
    }
    if (trailingZeroes(n) == K) {
        res++;
    }
}
return res;
```

前文 二分查找如何运用 说过，**对于这种具有单调性的函数，用 for 循环遍历，可以用二分查找进行降维打击**，对吧？

搜索有多少个 `n` 满足 `trailingZeroes(n) == K`，其实就是在问，**满足条件的 `n` 最小是多少，最大是多少，最大值和最小值一减，就可以算出来有多少个 `n` 满足条件了**，对吧？那不就是 二分查找 中「搜索左侧边界」和「搜索右侧边界」这两个事儿嘛？

先不急写代码，因为二分查找需要给一个搜索区间，也就是上界和下界，上述伪码中 `n` 的下界显然是 0，但上界是 `+inf`，这个正无穷应该如何表示出来呢？

首先，数学上的正无穷肯定是无法编程表示出来的，我们一般的方法是用一个非常大的值，大到这个值一定不会被取到。比如说 int 类型的最大值 `INT_MAX`（2^31 - 1，大约 31 亿），还不够的话就 long 类型的最大值 `LONG_MAX`（2^63 - 1，这个值就大到离谱了）。

那么我怎么知道需要多大才能「一定不会被取到」呢？**这就需要认真读题，看看题目给的数据范围有多大。**

这道题目实际上给了限制，`K` 是在 `[0, 10^9]` 区间内的整数，也就是说，`trailingZeroes(n)` 的结果最多可以达到 `10^9`。

然后我们可以反推，当 `trailingZeroes(n)` 结果为 `10^9` 时，`n` 为多少？这个不需要你精确计算出来，你只要找到一个数 `hi`，使得 `trailingZeroes(hi)` 比 `10^9` 大，就可以把 `hi` 当做正无穷，作为搜索区间的上界。

刚才说了，`trailingZeroes` 函数是单调函数，那我们就可以猜，先算一下 `trailingZeroes(INT_MAX)` 的结果，比 `10^9` 小一些，那再用 `LONG_MAX` 算一下，远超 `10^9` 了，所以 `LONG_MAX` 可以作为搜索的上界。

注意为了避免整型溢出的问题，`trailingZeroes` 函数需要把所有数据类型改成 long：

```
// 逻辑不变，数据类型全部改成 long
long trailingZeroes(long n) {
    long res = 0;
    for (long d = n; d / 5 > 0; d = d / 5) {
        res += d / 5;
    }
    return res;
}
```

现在就明确了问题：

在区间 `[0, LONG_MAX]` 中寻找满足 `trailingZeroes(n) == K` 的左侧边界和右侧边界。

根据前文 二分查找算法框架，可以直接把搜索左侧边界和右侧边界的框架 copy 过来：

```
/* 主函数 */
public int preimageSizeFZF(int K) {
    // 左边界和右边界之差 + 1 就是答案
    return (int)(right_bound(K) - left_bound(K) + 1);
}

/* 搜索 trailingZeroes(n) == K 的左侧边界 */
long left_bound(int target) {
    long lo = 0, hi = Long.MAX_VALUE;
    while (lo < hi) {
        long mid = lo + (hi - lo) / 2;
        if (trailingZeroes(mid) < target) {
            lo = mid + 1;
        } else if (trailingZeroes(mid) > target) {
            hi = mid;
        } else {
            hi = mid;
        }
    }
    return lo;
}

/* 搜索 trailingZeroes(n) == K 的右侧边界 */
long right_bound(int target) {
    long lo = 0, hi = Long.MAX_VALUE;
    while (lo < hi) {
        long mid = lo + (hi - lo) / 2;
        if (trailingZeroes(mid) < target) {
            lo = mid + 1;
        } else if (trailingZeroes(mid) > target) {
            hi = mid;
        } else {
            lo = mid + 1;
        }
```

```
        }
        return lo - 1;
    }
```

如果对二分查找的框架有任何疑问，建议好好复习一下前文 二分查找算法框架，这里就不展开了。

现在，这道题基本上就解决了，我们来分析一下它的时间复杂度吧。

时间复杂度主要是二分搜索，从数值上来说 `LONG_MAX` 是 2^63 - 1，大得离谱，但是二分搜索是对数级的复杂度，log(LONG_MAX) 是一个常数；每次二分的时候都会调用一次 `trailingZeroes` 函数，复杂度 O(logK)；所以总体的时间复杂度就是 O(logK)。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



如何在无限序列中随机抽取元素

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

382. 链表随机节点（中等）

398. 随机数索引（中等）

我最近在力扣上做到两道非常有意思的题目，382 和 398 题，关于水塘抽样算法（Reservoir Sampling），本质上是一种随机概率算法，解法应该说会者不难，难者不会。

我第一次见到这个算法问题是谷歌的一道算法题：给你一个**未知长度**的链表，请你设计一个算法，**只能遍历一次**，随机地返回链表中的一个节点。

这里说的随机是均匀随机（uniform random），也就是说，如果有 `n` 个元素，每个元素被选中的概率都是 `1/n`，不可以有统计意义上的偏差。

一般的想法就是，我先遍历一遍链表，得到链表的总长度 `n`，再生成一个 `[1,n]` 之间的随机数为索引，然后找到索引对应的节点，不就是一个随机的节点了吗？

但题目说了，只能遍历一次，意味着这种思路不可行。题目还可以再泛化，给一个未知长度的序列，如何在其中随机地选择 `k` 个元素？想要解决这个问题，就需要著名的水塘抽样算法了。

算法实现

先解决只抽取一个元素的问题，这个问题的难点在于，随机选择是「动态」的，比如说你现在你有 5 个元素，你已经随机选取了其中的某个元素 `a` 作为结果，但是现在再给你一个新元素 `b`，你应该留着 `a` 还是将 `b` 作为结果呢，以什么逻辑选择 `a` 和 `b` 呢，怎么证明你的选择方法在概率上是公平的呢？

先说结论，当你遇到第 `i` 个元素时，应该有 `1/i` 的概率选择该元素，`1 - 1/i` 的概率保持原有的选择。看代码容易理解这个思路：

```
/* 返回链表中一个随机节点的值 */
int getRandom(ListNode head) {
    Random r = new Random();
    int i = 0, res = 0;
    ListNode p = head;
    // while 循环遍历链表
```

```
        while (p != null) {
            i++;
            // 生成一个 [0, i) 之间的整数
            // 这个整数等于 0 的概率就是 1/i
            if (0 == r.nextInt(i)) {
                res = p.val;
            }
            p = p.next;
        }
        return res;
    }
```

对于概率算法，代码往往都是很浅显的，但是这种问题的关键在于证明，你的算法为什么是对的？为什么每次以 `1/i` 的概率更新结果就可以保证结果是平均随机（uniform random）？

证明：假设总共有 `n` 个元素，我们要的随机性无非就是每个元素被选择的概率都是 `1/n` 对吧，那么对于第 `i` 个元素，它被选择的概率就是：

$$
\begin{aligned}
&\frac{1}{i} \times (1 - \frac{1}{i+1}) \times (1 - \frac{1}{i+2}) \times \ldots \times (1 - \frac{1}{n}) \\
= &\frac{1}{i} \times \frac{i}{i+1} \times \frac{i+1}{i+2} \times \ldots \times \frac{n-1}{n} \\
= &\frac{1}{n}
\end{aligned}
$$

第 `i` 个元素被选择的概率是 `1/i`，第 `i+1` 次不被替换的概率是 `1 - 1/(i+1)`，以此类推，相乘就是第 `i` 个元素最终被选中的概率，就是 `1/n`。

因此，该算法的逻辑是正确的。

同理，如果要随机选择 `k` 个数，只要在第 `i` 个元素处以 `k/i` 的概率选择该元素，以 `1 - k/i` 的概率保持原有选择即可。代码如下：

```
/* 返回链表中 k 个随机节点的值 */
int[] getRandom(ListNode head, int k) {
    Random r = new Random();
    int[] res = new int[k];
    ListNode p = head;

    // 前 k 个元素先默认选上
    for (int j = 0; j < k && p != null; j++) {
        res[j] = p.val;
        p = p.next;
    }

    int i = k;
```

```
    // while 循环遍历链表
    while (p != null) {
        // 生成一个 [0, i) 之间的整数
        int j = r.nextInt(++i);
        // 这个整数小于 k 的概率就是 k/i
        if (j < k) {
            res[j] = p.val;
        }
        p = p.next;
    }
    return res;
}
```

对于数学证明，和上面区别不大：

$$
\begin{aligned}
&\frac{k}{i} \times (1 - \frac{k}{i+1} \times \frac{1}{k}) \times (1 - \frac{k}{i+2} \times \frac{1}{k}) \times \ldots \times (1 - \frac{k}{n} \times \frac{1}{k}) \\
&= \frac{k}{i} \times (1 - \frac{1}{i+1}) \times (1 - \frac{1}{i+2}) \times \ldots \times (1 - \frac{1}{n}) \\
&= \frac{k}{i} \times \frac{i}{i+1} \times \frac{i+1}{i+2} \times \ldots \times \frac{n-1}{n} \\
&= \frac{k}{n}
\end{aligned}
$$

因为虽然每次更新选择的概率增大了 `k` 倍，但是选到具体第 `i` 个元素的概率还是要乘 `1/k`，也就回到了上一个推导。

拓展延伸

以上的抽样算法时间复杂度是 O(n)，但不是最优的方法，更优化的算法基于几何分布（geometric distribution），时间复杂度为 O(k + klog(n/k))。由于涉及的数学知识比较多，这里就不列出了，有兴趣的读者可以自行搜索一下。

还有一种思路是基于「Fisher–Yates 洗牌算法」的。随机抽取 `k` 个元素，等价于对所有元素洗牌，然后选取前 `k` 个。只不过，洗牌算法需要对元素的随机访问，所以只能对数组这类支持随机存储的数据结构有效。

另外有一种思路也比较有启发意义：给每一个元素关联一个随机数，然后把每个元素插入一个容量为 `k` 的二叉堆（优先级队列）按照配对的随机数进行排序，最后剩下的 `k` 个元素也是随机的。

这个方案看起来似乎有点多此一举，因为插入二叉堆需要 O(logk) 的时间复杂度，所以整个抽样算法就需要 O(nlogk) 的复杂度，还不如我们最开始的算法。但是，这种思路可以指导我们解决**加权随机抽样算法**，权重越高，被随机选中的概率相应增大，这种情况在现实生活中是很常见的，比如你不往游戏里充钱，就永远抽不到皮肤。

最后，我想说随机算法虽然不多，但其实很有技巧的，读者不妨思考两个常见且看起来很简单的问题：

1、如何对带有权重的样本进行加权随机抽取？比如给你一个数组 `w`，每个元素 `w[i]` 代表权重，请你写一个算法，按照权重随机抽取索引。比如 `w = [1,99]`，算法抽到索引 0 的概率是 1%，抽到索引 1 的概率是

99%。

答案见 我的这篇文章。

2、实现一个生成器类，构造函数传入一个很长的数组，请你实现 `randomGet` 方法，每次调用随机返回数组中的一个元素，多次调用不能重复返回相同索引的元素。要求不能对该数组进行任何形式的修改，且操作的时间复杂度是 O(1)。

答案见 我的这篇文章。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜

labuladong公众号

东哥吃葡萄时竟吃出一道算法题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

吃葡萄

今天在牛客网上做了一道叫做「吃葡萄」的题目，非常有意思。

有三种葡萄，每种分别有 `a, b, c` 颗，现在有三个人，第一个人只吃第一种和第二种葡萄，第二个人只吃第二种和第三种葡萄，第三个人只吃第一种和第三种葡萄。

现在给你输入 `a, b, c` 三个值，请你适当安排，让三个人吃完所有的葡萄，算法返回**吃的最多的人最少要吃多少颗葡萄**。

题目链接：

https://www.nowcoder.com/questionTerminal/14c0359fb77a48319f0122ec175c9ada

牛客网的题目形式和力扣不一样，我去除输入和输出的处理，题目核心就是让你实现这样一个函数：

```
// 输入为三种葡萄的颗数，可能非常大，所以用 long 型
// 返回吃的最多的人最少要吃多少颗葡萄
long solution(long a, long b, long c);
```

题目解析

首先来理解一下题目，你怎么做到使得「吃得最多的那个人吃得最少」？

可以这样理解，**我们先不管每个人只能吃两种特定葡萄的约束**，你怎么让「吃得最多的那个人吃得最少」？

显然，只要平均分就行了，每个人吃 `(a+b+c)/3` 颗葡萄。即便不能整除，比如说 `a+b+c=8`，那也要尽可能平均分，就是说一个人吃 2 颗，另两个人吃 3 颗。

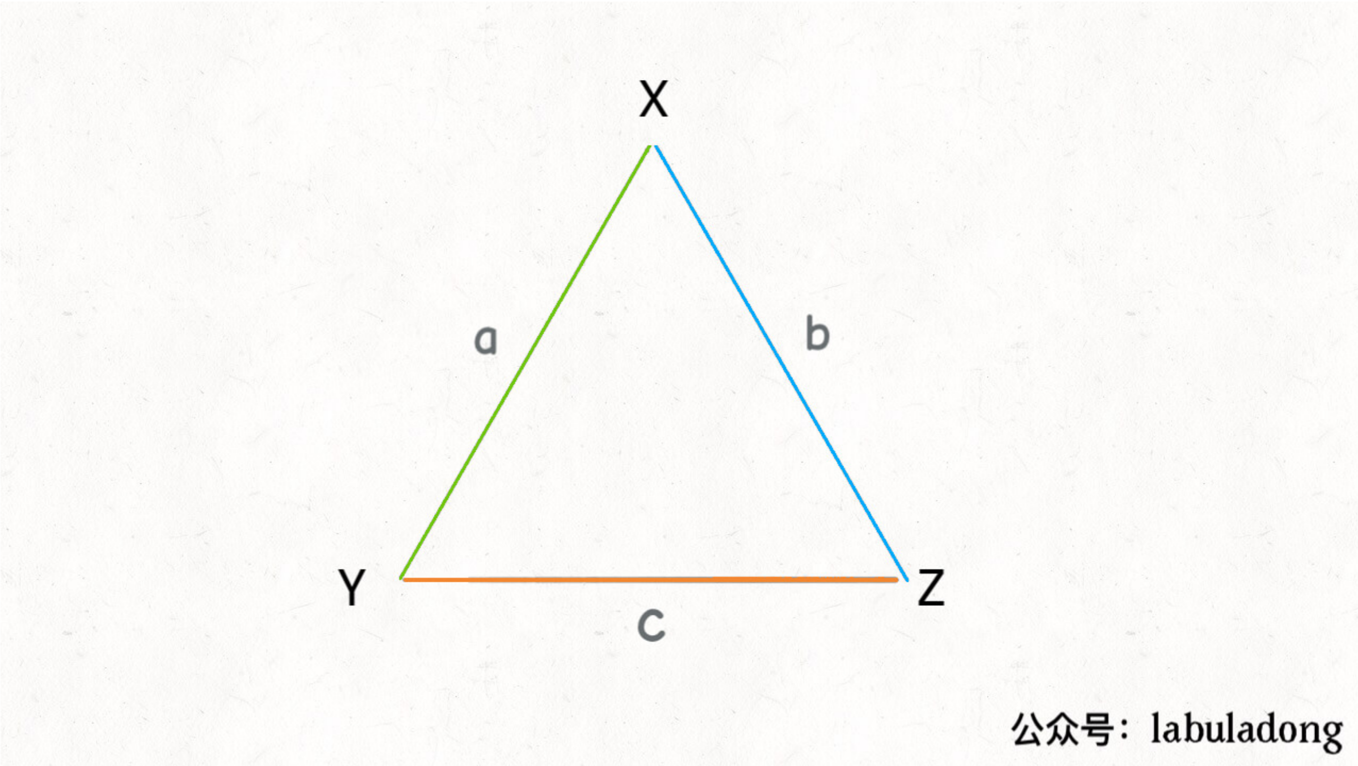
综上，「吃得最多的那个人吃得最少」就是让我们尽可能地平均分配，**而吃的最多的那个人吃掉的葡萄颗数就是 `(a+b+c)/3` 向上取整的结果，也就是 `(a+b+c+2)/3`**。

> PS：向上取整是一个常用的算法技巧。大部分编程语言中，如果你想计算 `M` 除以 `N`，`M / N` 会向下取整，你想向上取整的话，可以改成 `(M+(N-1)) / N`。

好了，刚才在讨论简单情况，现在考虑一下如果加上「每个人只能吃特定两种葡萄」的限制，怎么做？

也就是说，每个人只能吃特定两种葡萄，你也要**尽可能**给三个人平均分配，这样才能使得吃得最多的那个人吃得最少。

这可复杂了，如果用 X，Y，Z 表示这三个人，就会发现他们组成一个三角关系：



你让某一个人多吃某一种葡萄，就会产生连带效应，想着就头疼，这咋整？

思路分析

反正万事靠穷举呗，我一开始想了下回溯算法暴力穷举的可能性：

对于每一颗葡萄，可能被谁吃掉？有两种可能呗，那么我写一个回溯算法，把所有可能穷举出来，然后求个最值行不行？

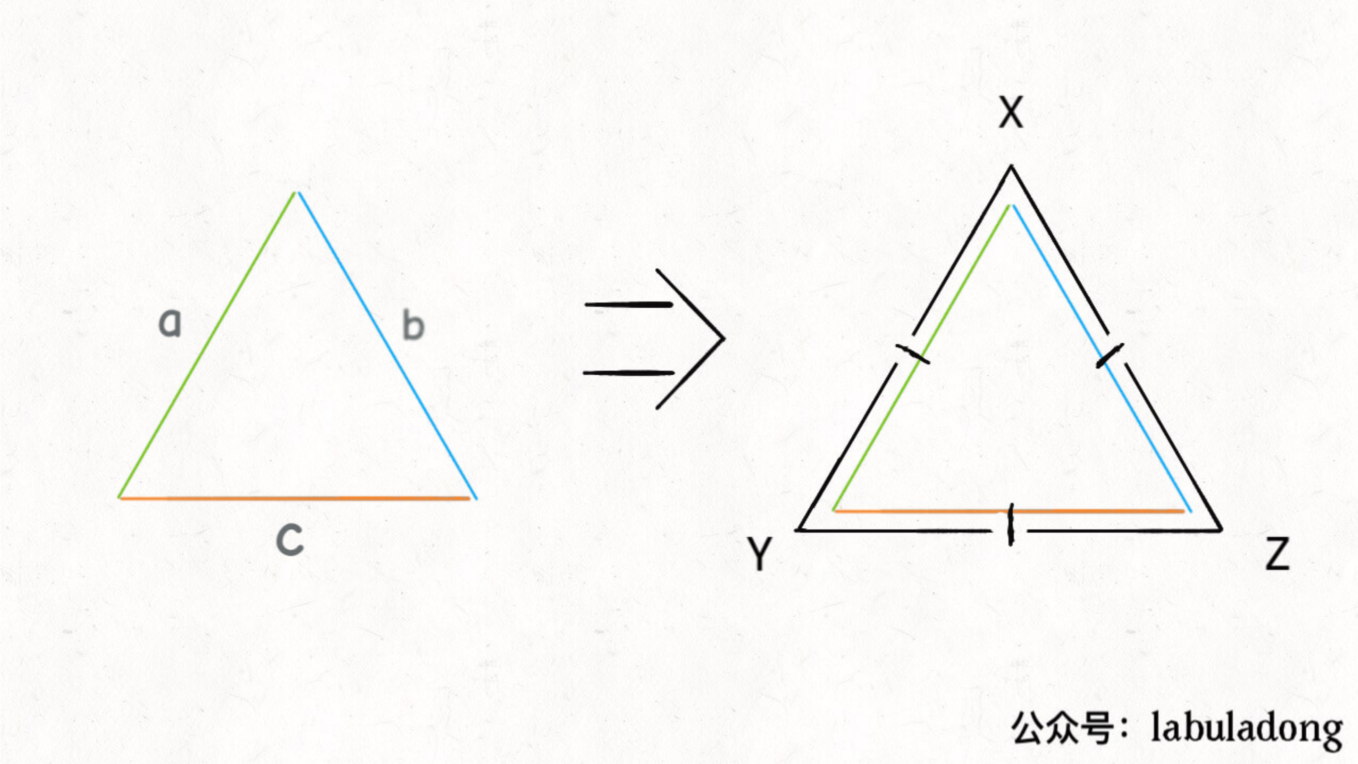
理论上是可行的，但是暴力算法的复杂度一般都是指数级，如果你以葡萄为「主角」进行穷举，看看变量 a，b，c 都是 long 型的数据，这个复杂度已经让我脊梁沟冒冷汗了。

那么这道题还是得取巧，**思路还是要回到如何「尽可能地平均分配」上面，那么事情就变得有意思起来**。

如果把葡萄的颗数 a，b，c 作为三条线段，它们的大小作为线段的长度，想一想它们可能组成什么几何图形？我们的目的是否可以转化成「尽可能平分这个几何图形的周长」？

三条线段组成的图形，那不就是三角形嘛？不急，我们小学就学过，三角形是要满足两边之和大于第三边的，假设 a < b < c，那么有下面两种情况：

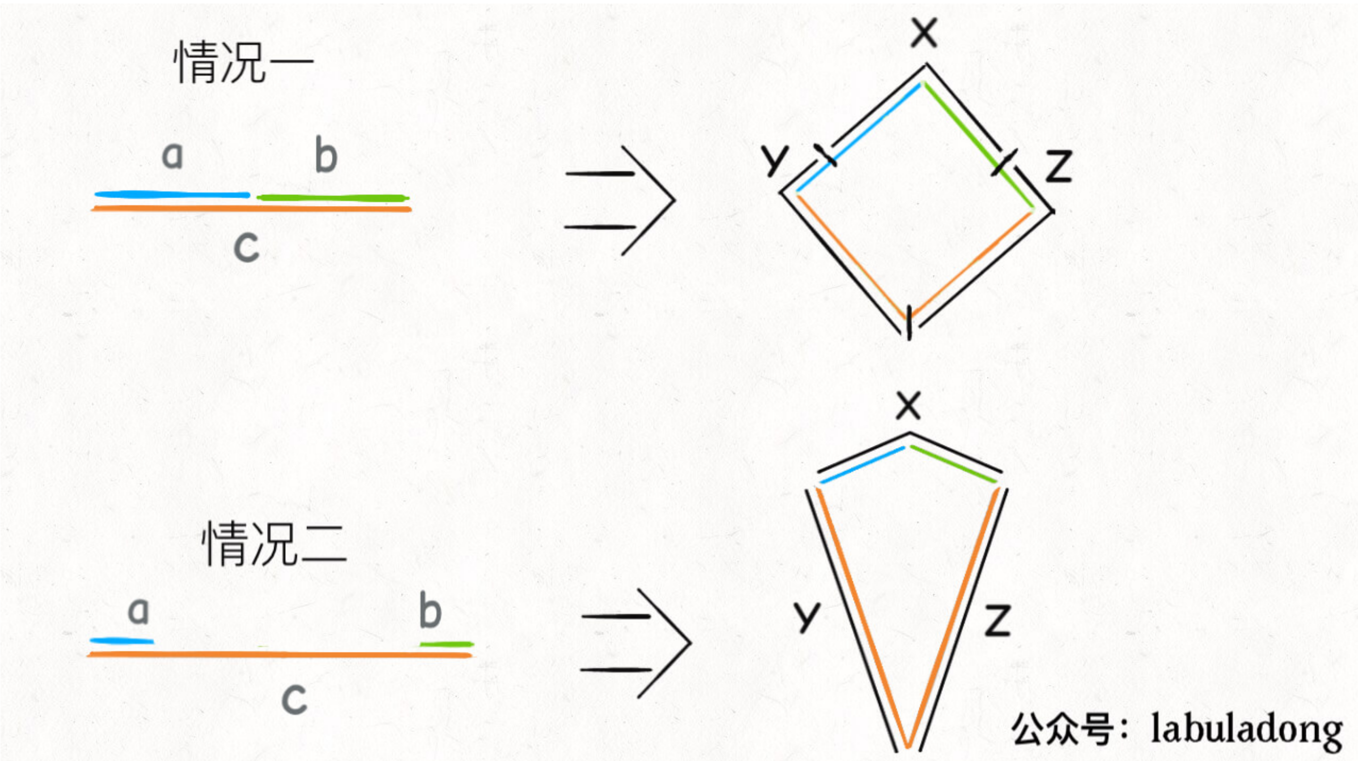
如果 a + b > c，那么可以构成一个三角形，只要把边 a，b，c 的中点画出来，这三点就一定可以把这个三角形的周长平分成三份，且每一份都包含两条边：



也就是说，这种情况下，三个人依然是可以平均分配所有葡萄的，吃的最多的人最少可以吃到的葡萄颗数依然是 `(a+b+c+2)/3`。

如果 `a + b <= c`，这三条边就不能组成一个封闭的图形了，那么我们可以将最长边 `c`「折断」，也就是形成一个四边形。

这里面有两种情况：



对于情况一，`a + b` 和 `c` 的差距还不大的时候，可以看到依然能够让三个人平分这个四边形，那么吃的最多的人最少可以吃到的葡萄颗数依然是 `(a+b+c+2)/3`。

随着 c 的不断增大，就会出现情况二，此时 c > 2*(a+b)，由于每个人口味的限制，为了尽可能平分，X 最多吃完 a 和 b，而 c 边需要被 Y 或 Z 平分，也就是说此时吃的最多的人最少可以吃到的葡萄颗数就是 (c+1)/2，即平分 c 边向上取整。

以上就是全部情况，翻译成代码如下：

```java
long solution(long a, long b, long c) {
    long[] nums = new long[]{a, b, c};
    Arrays.sort(nums);
    long sum = a + b + c;

    // 能够构成三角形，可完全平分
    if (nums[0] + nums[1] > nums[2]) {
        return (sum + 2) / 3;
    }
    // 不能构成三角形，平分最长边的情况
    if (2 * (nums[0] + nums[1]) < nums[2]) {
        return (nums[2] + 1) / 2;
    }
    // 不能构成三角形，但依然可以完全平分的情况
    return (sum + 2) / 3;
}
```

至此，这道题就被巧妙地解决了，时间复杂度仅需 O(1)，关键思路在于如何尽可能平分。

谁又能想到，吃个葡萄得借助几何图形？也许这就算法的魅力吧...

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

——————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



如何同时寻找缺失和重复的元素

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

645. 错误的集合（简单）

今天就聊一道很看起来简单却十分巧妙的问题，寻找缺失和重复的元素。之前的一篇文章 寻找缺失元素 也写过类似的问题，不过这次的和上次的问题使用的技巧不同。

这是力扣第 645 题「错误的集合」，我来描述一下这个题目：

给一个长度为 `N` 的数组 `nums`，其中本来装着 `[1..N]` 这 `N` 个元素，无序。但是现在出现了一些错误，`nums` 中的一个元素出现了重复，也就同时导致了另一个元素的缺失。请你写一个算法，找到 `nums` 中的重复元素和缺失元素的值。

```
// 返回两个数字，分别是 {dup, missing}
int[] findErrorNums(int[] nums);
```

比如说输入：`nums = [1,2,2,4]`，算法返回 `[2,3]`。

其实很容易解决这个问题，先遍历一次数组，用一个哈希表记录每个数字出现的次数，然后遍历一次 `[1..N]`，看看那个元素重复出现，那个元素没有出现，就 OK 了。

但问题是，这个常规解法需要一个哈希表，也就是 O(N) 的空间复杂度。你看题目给的条件那么巧，在 `[1..N]` 的几个数字中恰好有一个重复，一个缺失，**事出反常必有妖**，对吧。

O(N) 的时间复杂度遍历数组是无法避免的，所以我们可以想想办法如何降低空间复杂度，是否可以在 O(1) 的空间复杂度之下找到重复和缺失的元素呢？

思路分析

这个问题的特点是，每个元素和数组索引有一定的对应关系。

我们现在自己改造下问题，**暂且将 `nums` 中的元素变为 `[0..N-1]`，这样每个元素就和一个数组索引完全对应了，这样方便理解一些。**

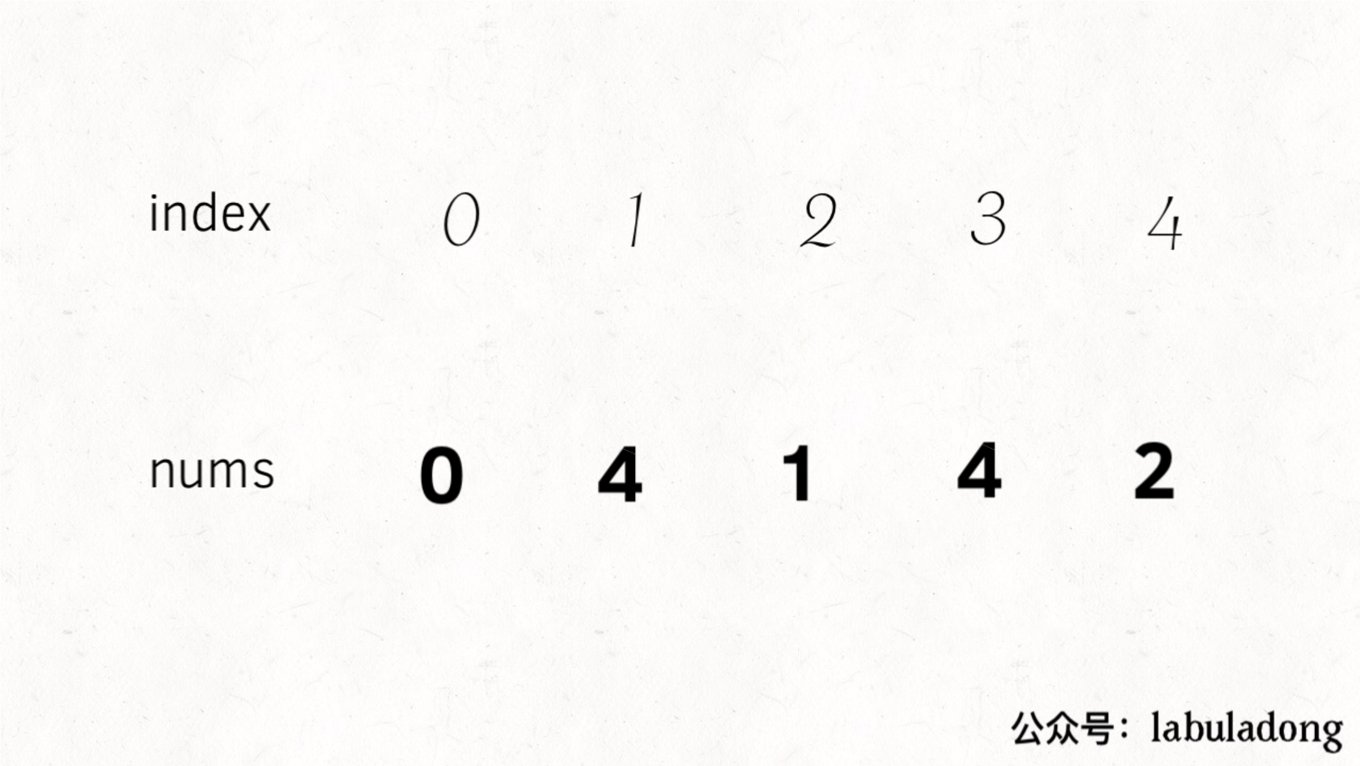
如果说 `nums` 中不存在重复元素和缺失元素，那么每个元素就和唯一一个索引值对应，对吧？

现在的问题是，有一个元素重复了，同时导致一个元素缺失了，这会产生什么现象呢？**会导致有两个元素对应到了同一个索引，而且会有一个索引没有元素对应过去。**

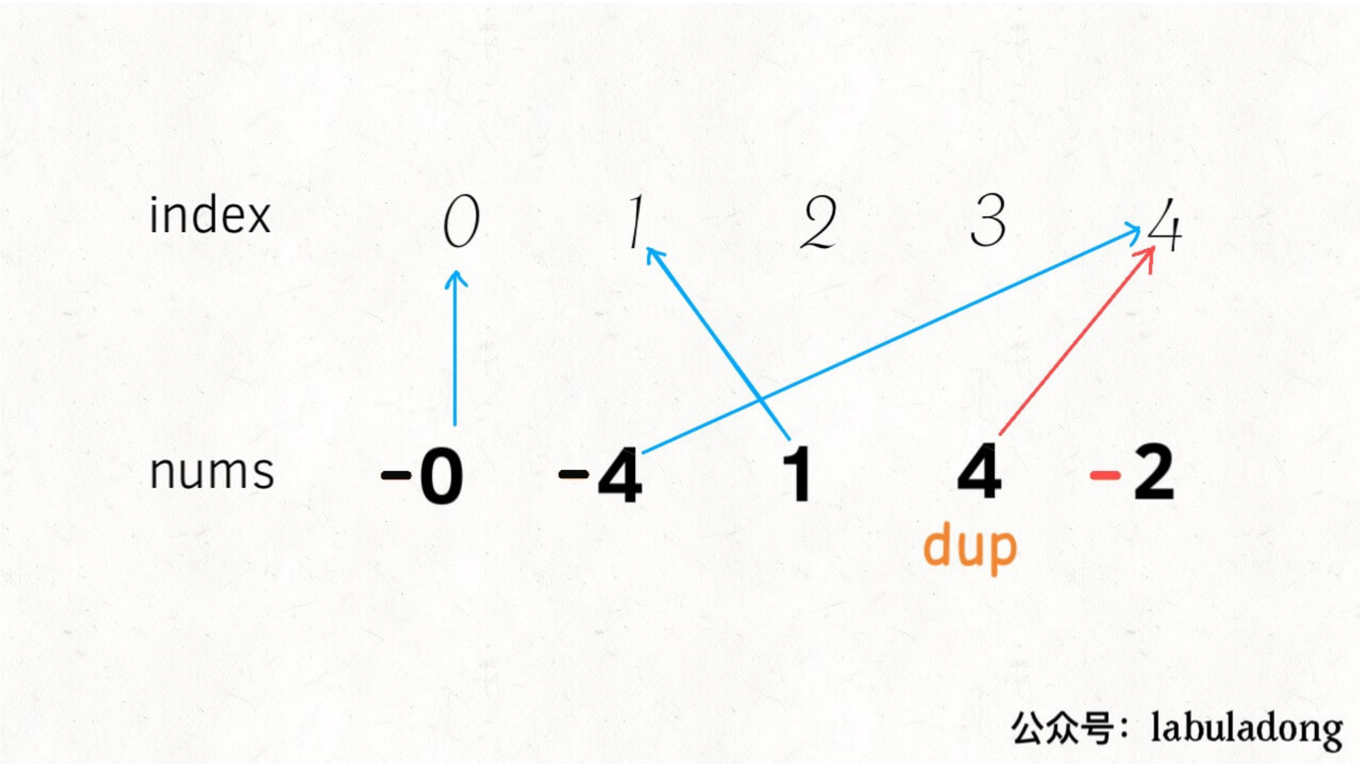
那么，如果我能够通过某些方法，找到这个重复对应的索引，不就是找到了那个重复元素么？找到那个没有元素对应的索引，不就是找到了那个缺失的元素了么？

那么，如何不使用额外空间判断某个索引有多少个元素对应呢？这就是这个问题的精妙之处了：

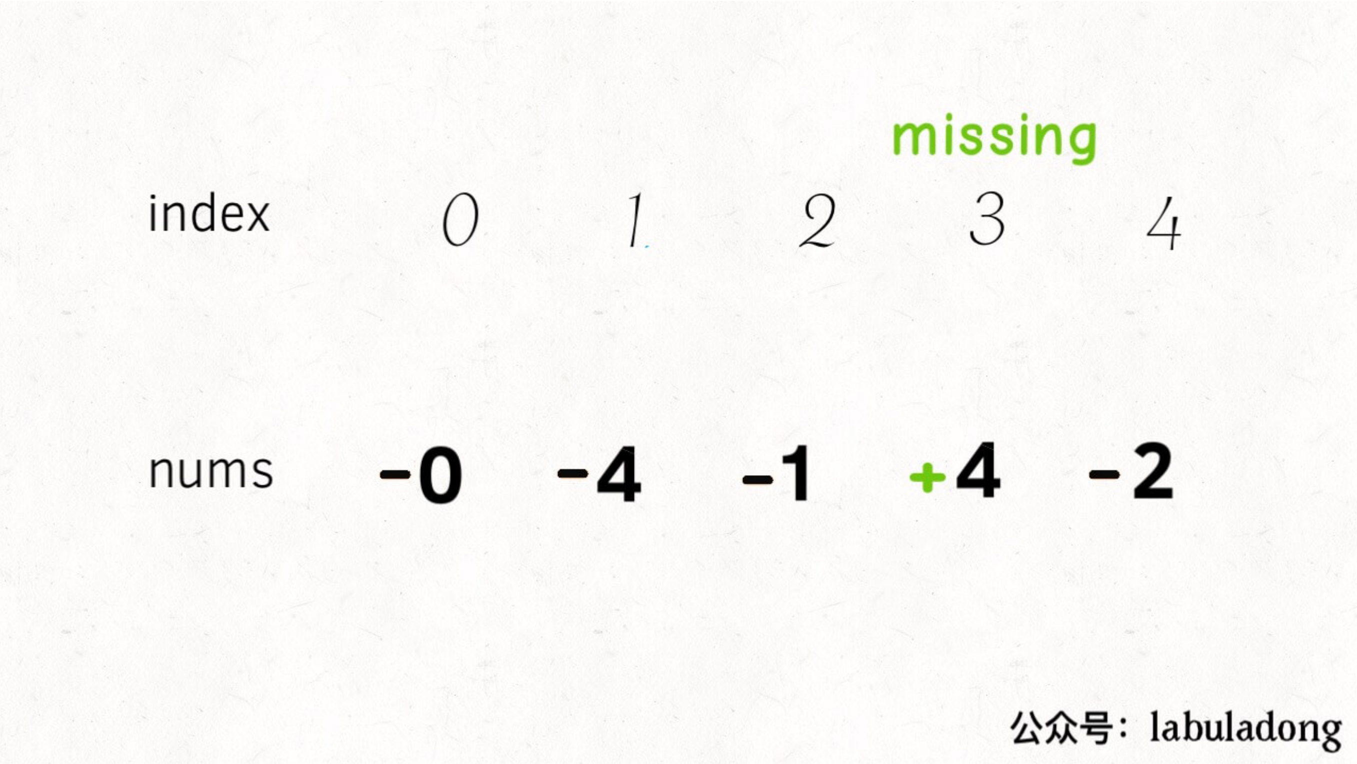
通过将每个索引对应的元素变成负数，以表示这个索引被对应过一次了：



如果出现重复元素 4，直观结果就是，索引 4 所对应的元素已经是负数了：



对于缺失元素 3，直观结果就是，索引 3 所对应的元素是正数：



对于这个现象，我们就可以翻译成代码了：

```
int[] findErrorNums(int[] nums) {
    int n = nums.length;
    int dup = -1;
    for (int i = 0; i < n; i++) {
        int index = Math.abs(nums[i]);
        // nums[index] 小于 0 则说明重复访问
        if (nums[index] < 0)
            dup = Math.abs(nums[i]);
        else
            nums[index] *= -1;
    }

    int missing = -1;
    for (int i = 0; i < n; i++)
        // nums[i] 大于 0 则说明没有访问
        if (nums[i] > 0)
            missing = i;

    return new int[]{dup, missing};
}
```

这个问题就基本解决了，别忘了我们刚才为了方便分析，假设元素是 [0..N-1]，但题目要求是 [1..N]，所以只要简单修改两处地方即可得到原题的答案：

```java
int[] findErrorNums(int[] nums) {
    int n = nums.length;
    int dup = -1;
    for (int i = 0; i < n; i++) {
        // 现在的元素是从 1 开始的
        int index = Math.abs(nums[i]) - 1;
        if (nums[index] < 0)
            dup = Math.abs(nums[i]);
        else
            nums[index] *= -1;
    }

    int missing = -1;
    for (int i = 0; i < n; i++)
        if (nums[i] > 0)
            // 将索引转换成元素
            missing = i + 1;

    return new int[]{dup, missing};
}
```

其实，元素从 1 开始是有道理的，也必须从一个非零数开始。因为如果元素从 0 开始，那么 0 的相反数还是自己，所以如果数字 0 出现了重复或者缺失，算法就无法判断 0 是否被访问过。我们之前的假设只是为了简化题目，更通俗易懂。

最后总结

对于这种数组问题，**关键点在于元素和索引是成对儿出现的，常用的方法是排序、异或、映射。**

映射的思路就是我们刚才的分析，将每个索引和元素映射起来，通过正负号记录某个元素是否被映射。

排序的方法也很好理解，对于这个问题，可以想象如果元素都被从小到大排序，如果发现索引对应的元素如果不相符，就可以找到重复和缺失的元素。

异或运算也是常用的，因为异或性质 `a ^ a = 0, a ^ 0 = a`，如果将索引和元素同时异或，就可以消除成对儿的索引和元素，留下的就是重复或者缺失的元素。可以看看前文 寻找缺失元素，介绍过这种方法。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

5.3 面试必知必会



除了像动态规划、回溯算法这种容易归类的算法类型，还有很多其他算法题并没有特别明显的特征，或者很难将一系列题目抽象汇总到一个算法技巧之下。

对于这类问题，只能说多做多总结，增加对这些算法问题的积累。

一个方法团灭 nSum 问题

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

15. 三数之和（中等）

18. 四数之和（中等）

经常刷 LeetCode 的读者肯定知道鼎鼎有名的 twoSum 问题，我们旧文 twoSum 问题的核心思想 就对 twoSum 的几个变种做了解析。

但是除了 twoSum 问题，LeetCode 上面还有 3Sum，4Sum 问题，我估计以后出个 5Sum，6Sum 也不是不可能。

那么，对于这种问题有没有什么好办法用套路解决呢？

今天 labuladong 就由浅入深，层层推进，用一个函数来解决所有 nSum 类型的问题。

一、twoSum 问题

上篇文章 twoSum 问题的核心思想 写了力扣上的 2Sum 问题，题目要求返回的是索引，这里我来编一道 2Sum 题目：

如果假设输入一个数组 nums 和一个目标和 target，**请你返回 nums 中能够凑出 target 的两个元素的值**，比如输入 nums = [1,3,5,6], target = 9，那么算法返回两个元素 [3,6]。可以假设只有且仅有一对儿元素可以凑出 target。

我们可以先对 nums 排序，然后利用前文 双指针技巧 写过的左右双指针技巧，从两端相向而行就行了：

```
vector<int> twoSum(vector<int>& nums, int target) {
    // 先对数组排序
    sort(nums.begin(), nums.end());
    // 左右指针
    int lo = 0, hi = nums.size() - 1;
    while (lo < hi) {
        int sum = nums[lo] + nums[hi];
        // 根据 sum 和 target 的比较，移动左右指针
        if (sum < target) {
            lo++;
```

```
        } else if (sum > target) {
            hi--;
        } else if (sum == target) {
            return {lo, hi};
        }
    }
    return {};
}
```

这样就可以解决这个问题，不过 labuladong 要魔改一下题目，把这个题目变得更泛华，更困难一点。

题目告诉我们可以假设 `nums` 中有且只有一个答案，且需要我们返回对应元素的索引，现在修改这些条件：**`nums` 中可能有多对儿元素之和都等于 `target`，请你的算法返回所有和为 `target` 的元素对儿，其中不能出现重复。**

函数签名如下：

```
vector<vector<int>> twoSumTarget(vector<int>& nums, int target);
```

比如说输入为 `nums = [1,3,1,2,2,3], target = 4`，那么算法返回的结果就是：`[[1,3],[2,2]]`。

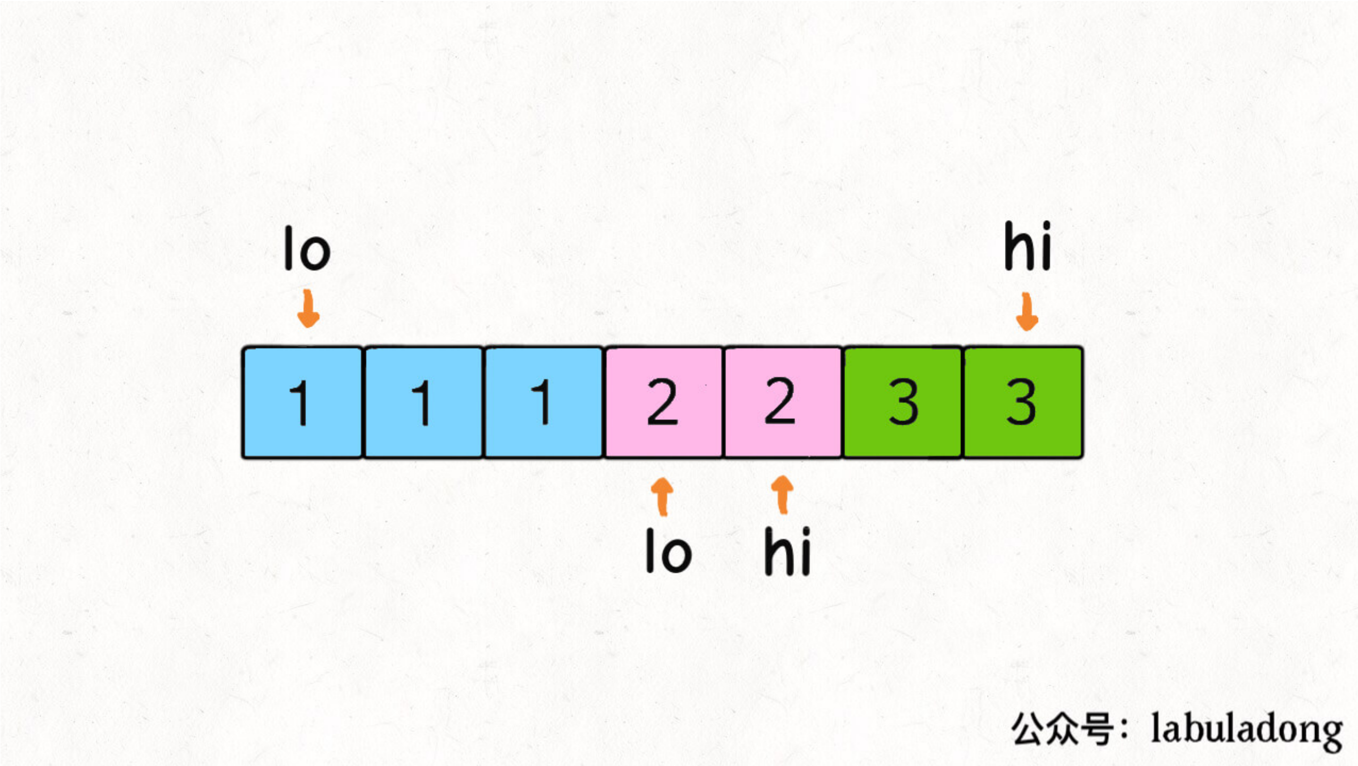
对于修改后的问题，返回元素的值而不是对应索引并没什么难度，关键难点是现在可能有多个和为 `target` 的数对儿，还不能重复，比如上述例子中 `[1,3]` 和 `[3,1]` 就算重复，只能算一次。

首先，基本思路肯定还是排序加双指针：

```
vector<vector<int>> twoSumTarget(vector<int>& nums, int target {
    // 先对数组排序
    sort(nums.begin(), nums.end());
    vector<vector<int>> res;
    int lo = 0, hi = nums.size() - 1;
    while (lo < hi) {
        int sum = nums[lo] + nums[hi];
        // 根据 sum 和 target 的比较，移动左右指针
        if      (sum < target) lo++;
        else if (sum > target) hi--;
        else {
            res.push_back({lo, hi});
            lo++; hi--;
        }
    }
    return res;
}
```

但是，这样实现会造成重复的结果，比如说 `nums = [1,1,1,2,2,3,3], target = 4`，得到的结果中 `[1,3]` 肯定会重复。

出问题的地方在于 `sum == target` 条件的 if 分支，当给 `res` 加入一次结果后，`lo` 和 `hi` 不应该只改变 1，而应该跳过所有重复的元素：



所以，可以对双指针的 while 循环做出如下修改：

```
while (lo < hi) {
    int sum = nums[lo] + nums[hi];
    // 记录索引 lo 和 hi 最初对应的值
    int left = nums[lo], right = nums[hi];
    if (sum < target)      lo++;
    else if (sum > target) hi--;
    else {
        res.push_back({left, right});
        // 跳过所有重复的元素
        while (lo < hi && nums[lo] == left) lo++;
        while (lo < hi && nums[hi] == right) hi--;
    }
}
```

这样就可以保证一个答案只被添加一次，重复的结果都会被跳过，可以得到正确的答案。不过，受这个思路的启发，其实前两个 if 分支也是可以做一点效率优化，跳过相同的元素：

```
vector<vector<int>> twoSumTarget(vector<int>& nums, int target) {
    // nums 数组必须有序
    sort(nums.begin(), nums.end());
    int lo = 0, hi = nums.size() - 1;
    vector<vector<int>> res;
    while (lo < hi) {
        int sum = nums[lo] + nums[hi];
```

```
            int left = nums[lo], right = nums[hi];
            if (sum < target) {
                while (lo < hi && nums[lo] == left) lo++;
            } else if (sum > target) {
                while (lo < hi && nums[hi] == right) hi--;
            } else {
                res.push_back({left, right});
                while (lo < hi && nums[lo] == left) lo++;
                while (lo < hi && nums[hi] == right) hi--;
            }
        }
        return res;
    }
```

这样，一个通用化的 `twoSum` 函数就写出来了，请确保你理解了该算法的逻辑，我们后面解决 `3Sum` 和 `4Sum` 的时候会复用这个函数。

这个函数的时间复杂度非常容易看出来，双指针操作的部分虽然有那么多 while 循环，但是时间复杂度还是 `O(N)`，而排序的时间复杂度是 `O(NlogN)`，所以这个函数的时间复杂度是 `O(NlogN)`。

二、3Sum 问题

这是 LeetCode 第 15 题：

15. 三数之和 labuladong 题解 思路

难度 中等 2295 收藏 分享 切换为英文 关注 反馈

给你一个包含 *n* 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 *a*，*b*，*c*，使得 $a + b + c = 0$？请你找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

```
给定数组 nums = [-1, 0, 1, 2, -1, -4]，

满足要求的三元组集合为：
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

题目就是让我们找 `nums` 中和为 0 的三个元素，返回所有可能的三元组（triple），函数签名如下：

```
vector<vector<int>> threeSum(vector<int>& nums);
```

这样，我们再泛化一下题目，不要光和为 0 的三元组了，计算和为 `target` 的三元组吧，同上面的 `twoSum` 一样，也不允许重复的结果：

——————————————————————

应合作方要求，本文不便在此发布，请扫码关注回复关键词「nsum」查看：

一个方法解决三道区间问题

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

1288. 删除被覆盖区间（中等）

56. 区间合并（中等）

986. 区间列表的交集（中等）

经常有读者问区间相关的问题，今天写一篇文章，秒杀三道区间相关的问题。

所谓区间问题，就是线段问题，让你合并所有线段、找出线段的交集等等。主要有两个技巧：

1、排序。常见的排序方法就是按照区间起点排序，或者先按照起点升序排序，若起点相同，则按照终点降序排序。当然，如果你非要按照终点排序，无非对称操作，本质都是一样的。

2、画图。就是说不要偷懒，勤动手，两个区间的相对位置到底有几种可能，不同的相对位置我们的代码应该怎么去处理。

废话不多说，下面我们来做题。

区间覆盖问题

这是力扣第 1288 题「删除被覆盖区间」，看下题目：

1288. 删除被覆盖区间 labuladong 题解 思路

难度 中等 14

给你一个区间列表，请你删除列表中被其他区间所覆盖的区间。

只有当 `c <= a` 且 `b <= d` 时，我们才认为区间 `[a,b)` 被左闭右开区间 `[c,d)` 覆盖。

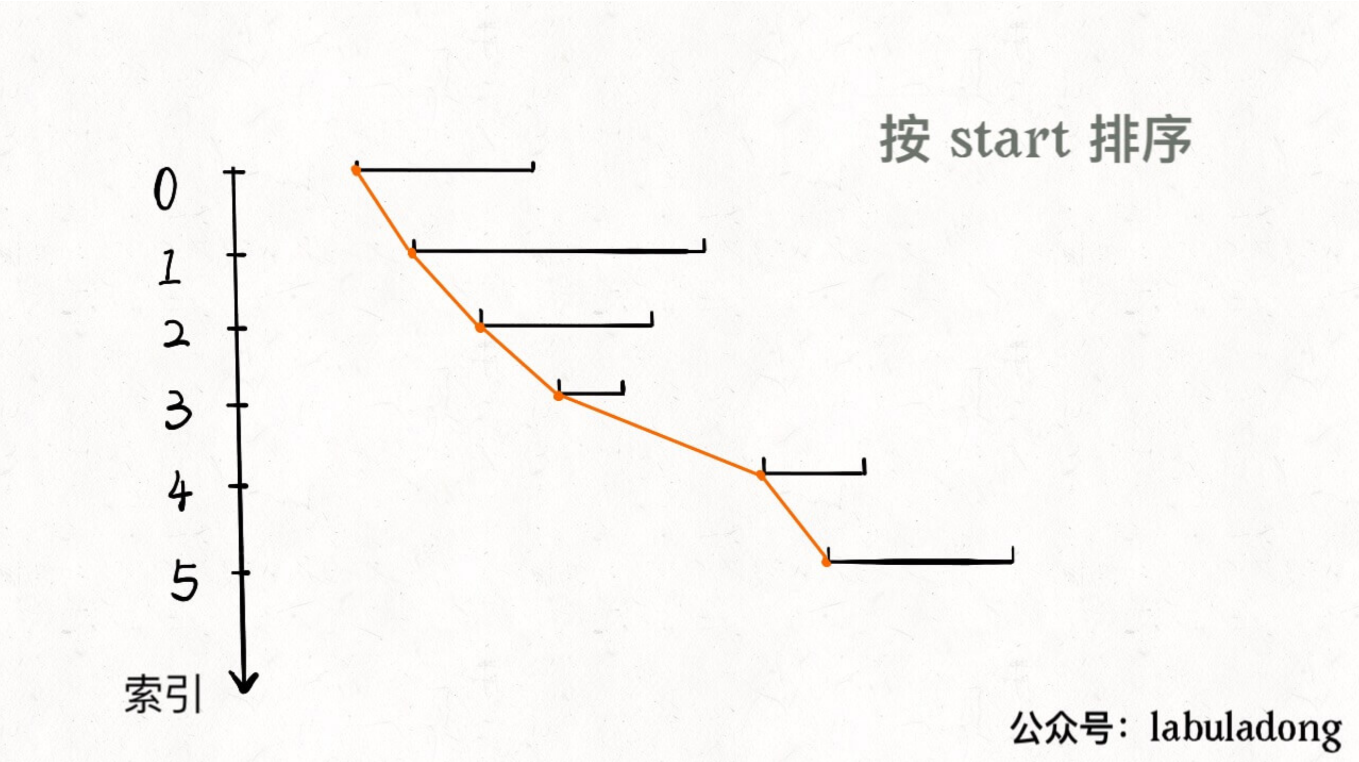
在完成所有删除操作后，请你返回列表中剩余区间的数目。

示例：

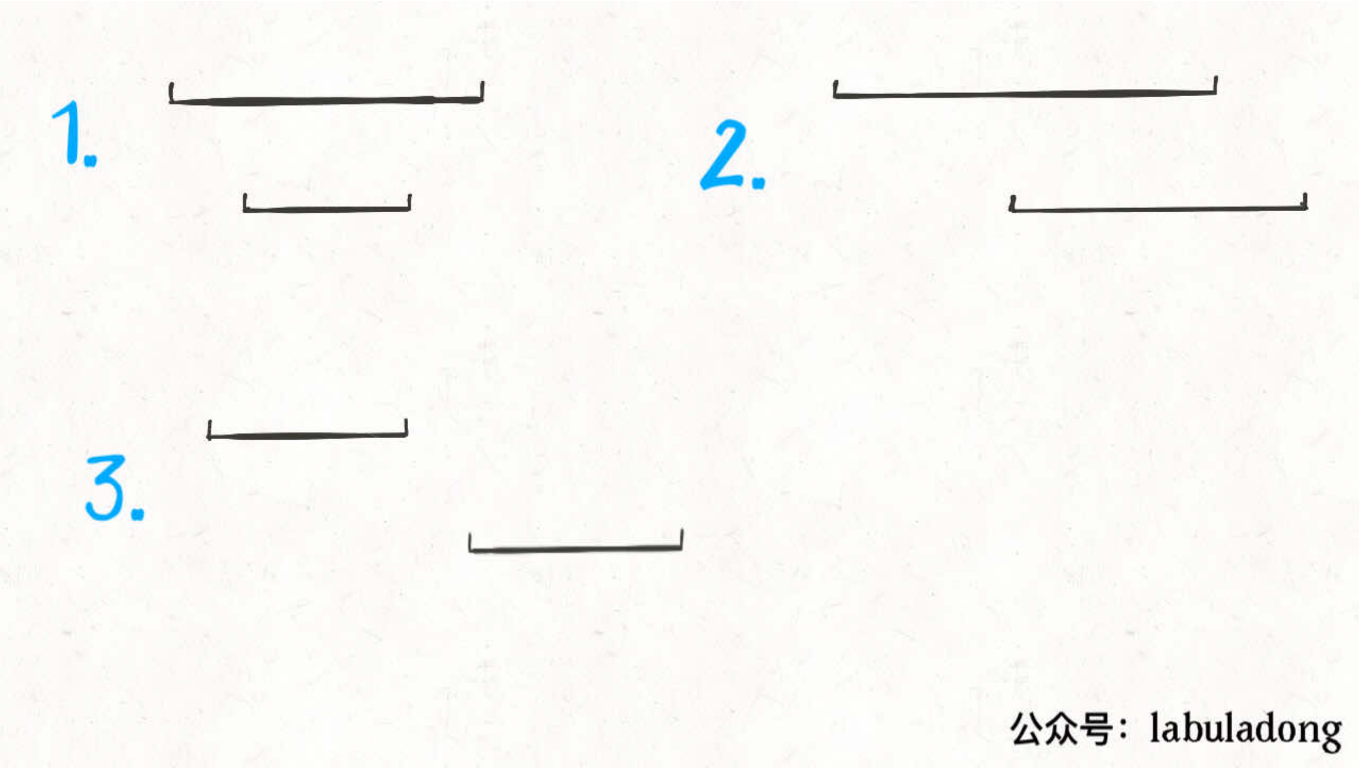
```
输入：intervals = [[1,4],[3,6],[2,8]]
输出：2
解释：区间 [3,6] 被区间 [2,8] 覆盖，所以它被删除了。
```

题目问我们，去除被覆盖区间之后，还剩下多少区间，**那么我们可以先算一算，被覆盖区间有多少个，然后和总数相减就是剩余区间数。**

对于这种区间问题，如果没啥头绪，首先排个序看看，比如我们按照区间的起点进行升序排序：



排序之后，两个相邻区间可能有如下三种相对位置：



对于这三种情况，我们应该这样处理：

对于情况一，找到了覆盖区间。

对于情况二，两个区间可以合并，成一个大区间。

对于情况三，两个区间完全不相交。

依据几种情况，我们可以写出如下代码：

```
int removeCoveredIntervals(int[][] intvs) {
    // 按照起点升序排列，起点相同时降序排列
    Arrays.sort(intvs, (a, b) -> {
        if (a[0] == b[0]) {
            return b[1] - a[1];
        }
        return a[0] - b[0];
    });

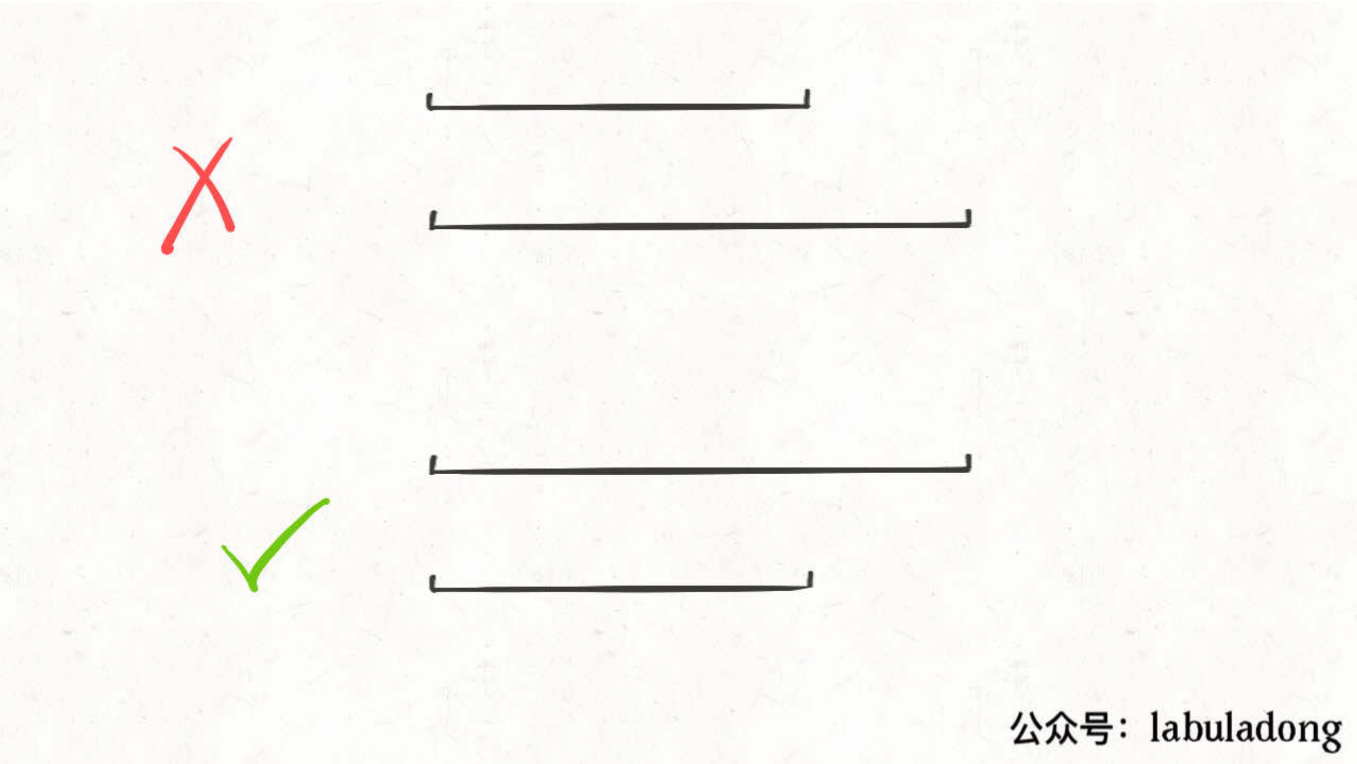
    // 记录合并区间的起点和终点
    int left = intvs[0][0];
    int right = intvs[0][1];

    int res = 0;
    for (int i = 1; i < intvs.length; i++) {
        int[] intv = intvs[i];
        // 情况一，找到覆盖区间
        if (left <= intv[0] && right >= intv[1]) {
            res++;
        }
```

```java
            // 情况二，找到相交区间，合并
            if (right >= intv[0] && right <= intv[1]) {
                right = intv[1];
            }
            // 情况三，完全不相交，更新起点和终点
            if (right < intv[0]) {
                left = intv[0];
                right = intv[1];
            }
        }

        return intvs.length - res;
    }
```

以上就是本题的解法代码，起点升序排列，终点降序排列的目的是防止如下情况：



对于这两个起点相同的区间，我们需要保证长的那个区间在上面（按照终点降序），这样才会被判定为覆盖，否则会被错误地判定为相交，少算一个覆盖区间。

区间合并问题

———————————

应合作方要求，本文不便在此发布，请扫码关注回复关键词「区间」查看：

快速排序亲兄弟：快速选择算法

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

215. 数组中的第 K 个最大元素（中等）

快速选择算法是一个非常经典的算法，和快速排序算法是亲兄弟。

原始题目很简单，给你输入一个无序的数组 `nums` 和一个正整数 `k`，让你计算 `nums` 中第 `k` 大的元素。

那你肯定说，给 `nums` 数组排个序，然后取第 `k` 个元素，也就是 `nums[k-1]`，不就行了吗？

当然可以，但是排序时间复杂度是 `O(NlogN)`，其中 `N` 表示数组 `nums` 的长度。

我们就想要第 `k` 大的元素，却给整个数组排序，有点杀鸡用牛刀的感觉，所以这里就有一些小技巧了，可以把时间复杂度降低到 `O(NlogK)` 甚至是 `O(N)`，下面我们就来具体讲讲。

力扣第 215 题「数组中的第 K 个最大元素」就是一道类似的题目，函数签名如下：

```
int findKthLargest(int[] nums, int k);
```

只不过题目要求找**第 `k` 个最大的元素**，和我们刚才说的**第 `k` 大的元素**在语义上不太一样，题目的意思相当于是把 `nums` 数组降序排列，然后返回第 `k` 个元素。

比如输入 `nums = [2,1,5,4], k = 2`，算法应该返回 4，因为 4 是 `nums` 中第 2 个**最大**的元素。

这种问题有两种解法，一种是二叉堆（优先队列）的解法，另一种就是标题说到的快速选择算法（Quick Select），我们分别来看。

解法一

二叉堆的解法比较简单，实际写算法题的时候，推荐大家写这种解法，先直接看代码吧：

```
int findKthLargest(int[] nums, int k) {
    // 小顶堆，堆顶是最小元素
    PriorityQueue<Integer>
        pq = new PriorityQueue<>();
```

```
    for (int e : nums) {
        // 每个元素都要过一遍二叉堆
        pq.offer(e);
        // 堆中元素多于 k 个时，删除堆顶元素
        if (pq.size() > k) {
            pq.poll();
        }
    }
    // pq 中剩下的是 nums 中 k 个最大元素，
    // 堆顶是最小的那个，即第 k 个最大元素
    return pq.peek();
}
```

二叉堆（优先队列）是比较常见的数据结构，可以认为它会自动排序，我们前文 手把手实现二叉堆数据结构 实现过这种结构，我就默认大家熟悉它的特性了。

看代码应该不难理解，可以把小顶堆 `pq` 理解成一个筛子，较大的元素会沉淀下去，较小的元素会浮上来；当堆大小超过 `k` 的时候，我们就删掉堆顶的元素，因为这些元素比较小，而我们想要的是前 `k` 个最大元素嘛。

当 `nums` 中的所有元素都过了一遍之后，筛子里面留下的就是最大的 `k` 个元素，而堆顶元素是堆中最小的元素，也就是「第 `k` 个最大的元素」。

二叉堆插入和删除的时间复杂度和堆中的元素个数有关，在这里我们堆的大小不会超过 `k`，所以插入和删除元素的复杂度是 `O(logK)`，再套一层 for 循环，总的时间复杂度就是 `O(NlogK)`。空间复杂度很显然就是二叉堆的大小，为 `O(K)`。

这个解法算是比较简单的吧，代码少也不容易出错，所以说如果笔试面试中出现类似的问题，建议用这种解法。唯一注意的是，Java 的 `PriorityQueue` 默认实现是小顶堆，有的语言的优先队列可能默认是大顶堆，可能需要做一些调整。

解法二

快速选择算法比较巧妙，时间复杂度更低，是快速排序的简化版，一定要熟悉思路。

我们先从快速排序讲起。

快速排序的逻辑是，若要对 `nums[lo..hi]` 进行排序，我们先找一个分界点 `p`，通过交换元素使得 `nums[lo..p-1]` 都小于等于 `nums[p]`，且 `nums[p+1..hi]` 都大于 `nums[p]`，然后递归地去 `nums[lo..p-1]` 和 `nums[p+1..hi]` 中寻找新的分界点，最后整个数组就被排序了。

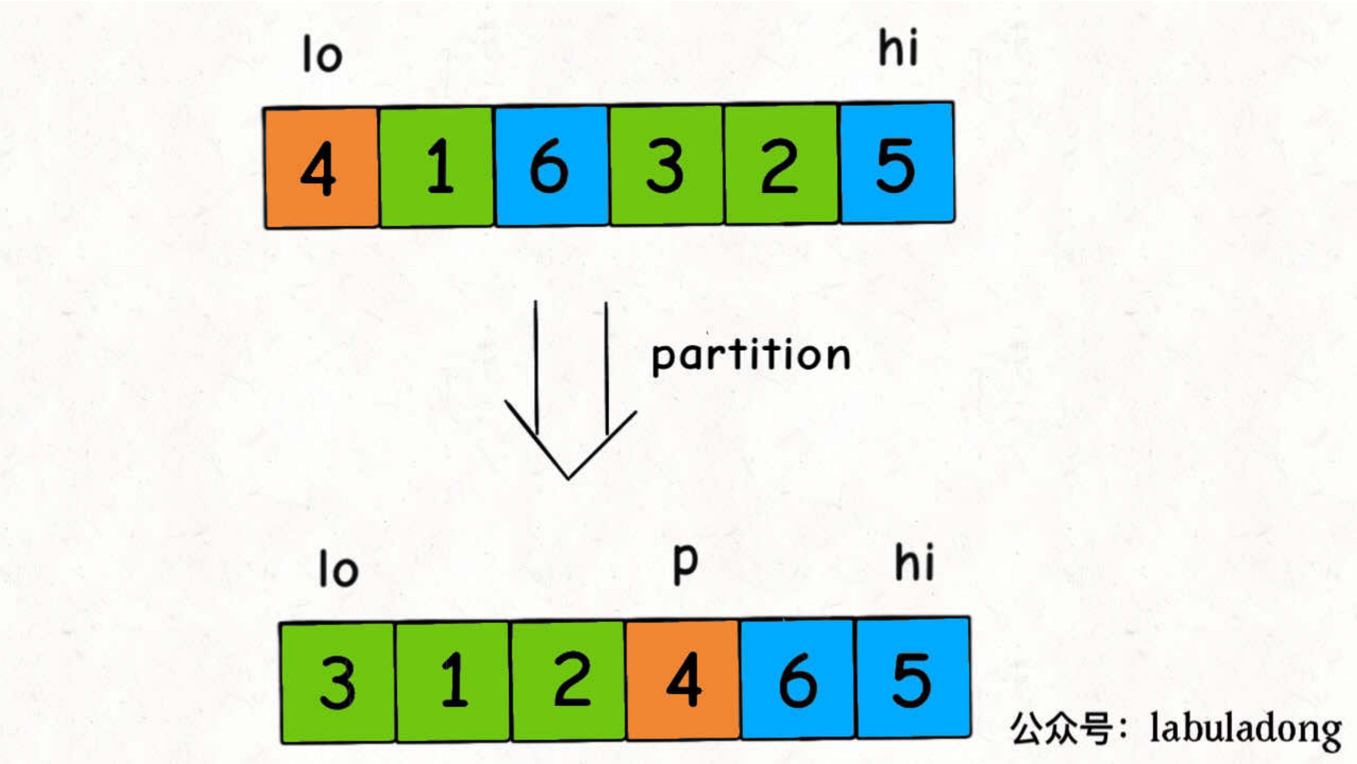
快速排序的代码如下：

```
/* 快速排序主函数 */
void sort(int[] nums) {
    // 一般要在这用洗牌算法将 nums 数组打乱，
    // 以保证较高的效率，我们暂时省略这个细节
    sort(nums, 0, nums.length - 1);
}

/* 快速排序核心逻辑 */
```

```java
void sort(int[] nums, int lo, int hi) {
    if (lo >= hi) return;
    // 通过交换元素构建分界点索引 p
    int p = partition(nums, lo, hi);
    // 现在 nums[lo..p-1] 都小于 nums[p],
    // 且 nums[p+1..hi] 都大于 nums[p]
    sort(nums, lo, p - 1);
    sort(nums, p + 1, hi);
}
```

关键就在于这个分界点索引 p 的确定，我们画个图看下 partition 函数有什么功效：



索引 p 左侧的元素都比 nums[p] 小，右侧的元素都比 nums[p] 大，意味着这个元素已经放到了正确的位置上，回顾快速排序的逻辑，递归调用会把 nums[p] 之外的元素也都放到正确的位置上，从而实现整个数组排序，这就是快速排序的核心逻辑。

那么这个 partition 函数如何实现的呢？看下代码：

```java
int partition(int[] nums, int lo, int hi) {
    if (lo == hi) return lo;
    // 将 nums[lo] 作为默认分界点 pivot
    int pivot = nums[lo];
    // j = hi + 1 因为 while 中会先执行 --
    int i = lo, j = hi + 1;
    while (true) {
        // 保证 nums[lo..i] 都小于 pivot
        while (nums[++i] < pivot) {
            if (i == hi) break;
        }
        // 保证 nums[j..hi] 都大于 pivot
```

```
            while (nums[--j] > pivot) {
                if (j == lo) break;
            }
            if (i >= j) break;
            // 如果走到这里，一定有：
            // nums[i] > pivot && nums[j] < pivot
            // 所以需要交换 nums[i] 和 nums[j]，
            // 保证 nums[lo..i] < pivot < nums[j..hi]
            swap(nums, i, j);
        }
        // 将 pivot 值交换到正确的位置
        swap(nums, j, lo);
        // 现在 nums[lo..j-1] < nums[j] < nums[j+1..hi]
        return j;
    }

    // 交换数组中的两个元素
    void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
```

熟悉快速排序逻辑的读者应该可以理解这段代码的含义了，这个 `partition` 函数细节较多，上述代码参考《算法4》，是众多写法中最漂亮简洁的一种，所以建议背住，这里就不展开解释了。

好了，对于快速排序的探讨到此结束，我们回到一开始的问题，寻找第 `k` 大的元素，和快速排序有什么关系？

注意这段代码：

```
int p = partition(nums, lo, hi);
```

我们刚说了，`partition` 函数会将 `nums[p]` 排到正确的位置，使得 `nums[lo..p-1] < nums[p] < nums[p+1..hi]`。

那么我们可以把 `p` 和 `k` 进行比较，如果 `p < k` 说明第 `k` 大的元素在 `nums[p+1..hi]` 中，如果 `p > k` 说明第 `k` 大的元素在 `nums[lo..p-1]` 中。

所以我们可以复用 `partition` 函数来实现这道题目，不过在这之前还是要做一下索引转化：

题目要求的是「第 `k` 个最大元素」，这个元素其实就是 `nums` 升序排序后「索引」为 `len(nums) - k` 的这个元素。

这样就可以写出解法代码：

```
int findKthLargest(int[] nums, int k) {
    int lo = 0, hi = nums.length - 1;
    // 索引转化
```

```
    k = nums.length - k;
    while (lo <= hi) {
        // 在 nums[lo..hi] 中选一个分界点
        int p = partition(nums, lo, hi);
        if (p < k) {
            // 第 k 大的元素在 nums[p+1..hi] 中
            lo = p + 1;
        } else if (p > k) {
            // 第 k 大的元素在 nums[lo..p-1] 中
            hi = p - 1;
        } else {
            // 找到第 k 大元素
            return nums[p];
        }
    }
    return -1;
}
```

这个代码框架其实非常像我们前文 二分搜索框架 的代码，这也是这个算法高效的原因，但是时间复杂度为什么是 O(N) 呢？按理说类似二分搜索的逻辑，时间复杂度应该一定会出现对数才对呀？

其实这个 O(N) 的时间复杂度是个均摊复杂度，因为我们的 partition 函数中需要利用 双指针技巧 遍历 nums[lo..hi]，那么总共遍历了多少元素呢？

最好情况下，每次 p 都恰好是正中间 (lo + hi) / 2，那么遍历的元素总数就是：

N + N/2 + N/4 + N/8 + ... + 1

这就是等比数列求和公式嘛，求个极限就等于 2N，所以遍历元素个数为 2N，时间复杂度为 O(N)。

但我们其实不能保证每次 p 都是正中间的索引的，最坏情况下 p 一直都是 lo + 1 或者一直都是 hi - 1，遍历的元素总数就是：

N + (N - 1) + (N - 2) + ... + 1

这就是个等差数列求和，时间复杂度会退化到 O(N^2)，**为了尽可能防止极端情况发生，我们需要在算法开始的时候对 nums 数组来一次随机打乱**：

```
int findKthLargest(int[] nums, int k) {
    // 首先随机打乱数组
    shuffle(nums);
    // 其他都不变
    int lo = 0, hi = nums.length - 1;
    k = nums.length - k;
    while (lo <= hi) {
        // ...
    }
    return -1;
}

// 对数组元素进行随机打乱
void shuffle(int[] nums) {
```

```java
        int n = nums.length;
        Random rand = new Random();
        for (int i = 0 ; i < n; i++) {
            // 从 i 到最后随机选一个元素
            int r = i + rand.nextInt(n - i);
            swap(nums, i, r);
        }
    }
```

前文 洗牌算法详解 写过随机乱置算法，这里就不展开了。当你加上这段代码之后，平均时间复杂度就是 `O(N)` 了，提交代码后运行速度大幅提升。

总结一下，快速选择算法就是快速排序的简化版，复用了 `partition` 函数，快速定位第 `k` 大的元素。相当于对数组部分排序而不需要完全排序，从而提高算法效率，将平均时间复杂度降到 `O(N)`。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜

labuladong公众号

分治算法详解：运算优先级

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

241. 为运算表达式设计优先级（中等）

我们号已经写了 动态规划算法，回溯（DFS）算法，BFS 算法，贪心算法，双指针算法，滑动窗口算法，现在就差个分治算法没写了，今天来写一下。

其实，我觉得回溯、分治和动态规划算法可以划为一类，因为它们都会涉及递归。

回溯算法就一种简单粗暴的算法技巧，说白了就是一个暴力穷举算法，比如让你 用回溯算法求子集、全排列、组合，你就穷举呗，就考你会不会漏掉或者多算某些情况。

动态规划是一类算法问题，肯定是让你求最值的。因为动态规划问题拥有 最优子结构，可以通过状态转移方程从小规模的子问题最优解推导出大规模问题的最优解。

分治算法呢，可以认为是一种算法思想，通过将原问题分解成小规模的子问题，然后根据子问题的结果构造出原问题的答案。这里有点类似动态规划，所以说运用分治算法也需要满足一些条件，你的原问题结果应该可以通过合并子问题结果来计算。

其实这几个算法之间界定并没有那么清晰，有时候回溯算法加个备忘录似乎就成动态规划了，而分治算法有时候也可以加备忘录进行剪枝。

我觉得吧，没必要过分纠结每个算法的定义，定义这东西无非文学词汇而已，反正能把题做出来你说这是啥算法都行，**所以大家还是得多刷题，刷出感觉，各种算法都手到擒来。**

最典型的分治算法就是归并排序了，核心逻辑如下：

```
void sort(int[] nums, int lo, int hi) {
    int mid = (lo + hi) / 2;
    /****** 分 ******/
    // 对数组的两部分分别排序
    sort(nums, lo, mid);
    sort(nums, mid + 1, hi);
    /****** 治 ******/
    // 合并两个排好序的子数组
    merge(nums, lo, mid, hi);
}
```

「对数组排序」是一个可以运用分治思想的算法问题，只要我先把数组的左半部分排序，再把右半部分排序，最后把两部分合并，不就是对整个数组排序了吗？

下面来看一道具体的算法题。

添加括号的所有方式

我来借力扣第 241 题「为运算表达式设计优先级」来讲讲什么是分治算法，先看看题目：

241. 为运算表达式设计优先级 labuladong 题解 思路

难度 中等 285

给定一个含有数字和运算符的字符串，为表达式添加括号，改变其运算优先级以求出不同的结果。你需要给出所有可能的组合的结果。有效的运算符号包含 `+`, `-` 以及 `*` 。

示例:

```
输入: "2*3-4*5"
输出: [-34, -14, -10, -10, 10]
解释:
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10
```

简单说，就是给你输入一个算式，你可以给它随意加括号，**请你穷举出所有可能的加括号方式，并计算出对应的结果。**

函数签名如下：

```java
// 计算所有加括号的结果
List<Integer> diffWaysToCompute(String input);
```

看到这道题的第一感觉肯定是复杂，我要穷举出所有可能的加括号方式，是不是还要考虑括号的合法性？是不是还要考虑计算的优先级？

是的，这些都要考虑，但是不需要我们来考虑。利用分治思想和递归函数，算法会帮我们考虑一切细节，也许这就是算法的魅力吧，哈哈哈。

废话不多说，解决本题的关键有两点：

1、不要思考整体，而是把目光聚焦局部，只看一个运算符。

这一点我们前文经常提及，比如 手把手刷二叉树第一期 就告诉你解决二叉树系列问题只要思考每个节点需要做什么，而不要思考整棵树需要做什么。

说白了，解决递归相关的算法问题，就是一个化整为零的过程，你必须瞄准一个小的突破口，然后把问题拆解，大而化小，利用递归函数来解决。

2、明确递归函数的定义是什么，相信并且利用好函数的定义。

这也是前文经常提到的一个点，因为递归函数要自己调用自己，你必须搞清楚函数到底能干嘛，才能正确进行递归调用。

下面来具体解释下这两个关键点怎么理解。

我们先举个例子，比如我给你输入这样一个算式：

`1 + 2 * 3 - 4 * 5`

请问，这个算式有几种加括号的方式？请在一秒之内回答我。

估计你回答不出来，因为括号可以嵌套，要穷举出来肯定得费点功夫。

不过呢，嵌套这个事情吧，我们人类来看是很头疼的，但对于算法来说嵌套括号不要太简单，一次递归就可以嵌套一层，一次搞不定大不了多递归几次。

所以，作为写算法的人类，我们只需要思考，如果不让括号嵌套（即只加一层括号），有几种加括号的方式？

还是上面的例子，显然我们有四种加括号方式：

`(1) + (2 * 3 - 4 * 5)`

`(1 + 2) * (3 - 4 * 5)`

`(1 + 2 * 3) - (4 * 5)`

`(1 + 2 * 3 - 4) * (5)`

发现规律了么？**其实就是按照运算符进行分割，给每个运算符的左右两部分加括号**，这就是之前说的第一个关键点，不要考虑整体，而是聚焦每个运算符。

现在单独说上面的第三种情况：

`(1 + 2 * 3) - (4 * 5)`

我们用减号 `-` 作为分隔，把原算式分解成两个算式 `1 + 2 * 3` 和 `4 * 5`。

分治分治，分而治之，**这一步就是把原问题进行了「分」，我们现在要开始「治」了。**

`1 + 2 * 3` 可以有两种加括号的方式，分别是：

`(1) + (2 * 3) = 7`

`(1 + 2) * (3) = 9`

或者我们可以写成这种形式：

`1 + 2 * 3 = [9, 7]`

而 `4 * 5` 当然只有一种加括号方式，就是 `4 * 5 = [20]`。

然后呢，你能不能通过上述结果推导出 `(1 + 2 * 3) - (4 * 5)` 有几种加括号方式，或者说有几种不同的结果？

显然，可以推导出来 `(1 + 2 * 3) - (4 * 5)` 有两种结果，分别是：

`9 - 20 = -11`

`7 - 20 = -13`

那你可能要问了， `1 + 2 * 3 = [9, 7]` 的结果是我们自己看出来的，如何让算法计算出来这个结果呢？

这个简单啊，再回头看下题目给出的函数签名：

```
// 定义：计算算式 input 所有可能的运算结果
List<Integer> diffWaysToCompute(String input);
```

这个函数不就是干这个事儿的吗？**这就是我们之前说的第二个关键点，明确函数的定义，相信并且利用这个函数定义。**

你甭管这个函数怎么做到的，你相信它能做到，然后用就行了，最后它就真的能做到了。

那么，对于 `(1 + 2 * 3) - (4 * 5)` 这个例子，我们的计算逻辑其实就是这段代码：

```
List<Integer> diffWaysToCompute("(1 + 2 * 3) - (4 * 5)") {
    List<Integer> res = new LinkedList<>();
    /****** 分 ******/
    List<Integer> left = diffWaysToCompute("1 + 2 * 3");
    List<Integer> right = diffWaysToCompute("4 * 5");
    /****** 治 ******/
    for (int a : left)
        for (int b : right)
            res.add(a - b);

    return res;
}
```

好，现在 `(1 + 2 * 3) - (4 * 5)` 这个例子是如何计算的，你应该完全理解了吧，那么回来看我们的原始问题。

原问题 `1 + 2 * 3 - 4 * 5` 是不是只有 `(1 + 2 * 3) - (4 * 5)` 这一种情况？是不是只能从减号 `-` 进行分割？

不是，每个运算符都可以把原问题分割成两个子问题，刚才已经列出了所有可能的分割方式：

(1) + (2 * 3 - 4 * 5)

(1 + 2) * (3 - 4 * 5)

(1 + 2 * 3) - (4 * 5)

(1 + 2 * 3 - 4) * (5)

所以，我们需要穷举上述的每一种情况，可以进一步细化一下解法代码：

```
List<Integer> diffWaysToCompute(String input) {
    List<Integer> res = new LinkedList<>();
    for (int i = 0; i < input.length(); i++) {
        char c = input.charAt(i);
        // 扫描算式 input 中的运算符
        if (c == '-' || c == '*' || c == '+') {
            /****** 分 ******/
            // 以运算符为中心，分割成两个字符串，分别递归计算
            List<Integer>
                left = diffWaysToCompute(input.substring(0, i));
            List<Integer>
                right = diffWaysToCompute(input.substring(i + 1));
            /****** 治 ******/
            // 通过子问题的结果，合成原问题的结果
            for (int a : left)
                for (int b : right)
                    if (c == '+')
                        res.add(a + b);
                    else if (c == '-')
                        res.add(a - b);
                    else if (c == '*')
                        res.add(a * b);
        }
    }
    // base case
    // 如果 res 为空，说明算式是一个数字，没有运算符
    if (res.isEmpty()) {
        res.add(Integer.parseInt(input));
    }
    return res;
}
```

有了刚才的铺垫，这段代码应该很好理解了吧，就是扫描输入的算式 input，每当遇到运算符就进行分割，递归计算出结果后，根据运算符来合并结果。

这就是典型的分治思路，先「分」后「治」，先按照运算符将原问题拆解成多个子问题，然后通过子问题的结果来合成原问题的结果。

当然，一个重点在这段代码：

```java
    // base case
    // 如果 res 为空，说明算式是一个数字，没有运算符
    if (res.isEmpty()) {
        res.add(Integer.parseInt(input));
    }
```

递归函数必须有个 base case 用来结束递归，其实这段代码就是我们分治算法的 base case，代表着你「分」到什么时候可以开始「治」。

我们是按照运算符进行「分」的，一直这么分下去，什么时候是个头？显然，当算式中不存在运算符的时候就可以结束了。

那为什么以 `res.isEmpty()` 作为判断条件？因为当算式中不存在运算符的时候，就不会触发 if 语句，也就不会给 `res` 中添加任何元素。

至此，这道题的解法代码就写出来了，但是时间复杂度是多少呢？

如果单看代码，真的很难通过 for 循环的次数看出复杂度是多少，所以我们需要改变思路，本题在求所有可能的计算结果，不就**相当于在求算式 `input` 的所有合法括号组合**吗？

那么，对于一个算式，有多少种合法的括号组合呢？这就是著名的「卡特兰数」了，最终结果是一个组合数，推导过程稍有些复杂，我这里就不写了，有兴趣的读者可以自行搜索了解一下。

其实本题还有一个小的优化，可以进行递归剪枝，减少一些重复计算，比如说输入的算式如下：

`1 + 1 + 1 + 1 + 1`

那么按照算法逻辑，按照运算符进行分割，一定存在下面两种分割情况：

`(1 + 1) + (1 + 1 + 1)`

`(1 + 1 + 1) + (1 + 1)`

算法会依次递归每一种情况，其实就是冗余计算嘛，所以我们可以对解法代码稍作修改，加一个备忘录来避免这种重复计算：

```java
// 备忘录
HashMap<String, List<Integer>> memo = new HashMap<>();

List<Integer> diffWaysToCompute(String input) {
    // 避免重复计算
    if (memo.containsKey(input)) {
        return memo.get(input);
    }
    /****** 其他都不变 ******/
    List<Integer> res = new LinkedList<>();
    for (int i = 0; i < input.length(); i++) {
        // ...
    }
    if (res.isEmpty()) {
        res.add(Integer.parseInt(input));
```

```
        }
        /***********************/

        // 将结果添加进备忘录
        memo.put(input, res);
        return res;
    }
```

当然，这个优化没有改变原始的复杂度，只是对一些特殊情况做了剪枝，提升了效率。

最后总结

解决上述算法题利用了分治思想，以每个运算符作为分割点，把复杂问题分解成小的子问题，递归求解子问题，然后再通过子问题的结果计算出原问题的结果。

把大规模的问题分解成小规模的问题递归求解，应该是计算机思维的精髓了吧，建议大家多练，如果本文对你有帮助，记得分享给你的朋友哦~

接下来可阅读：

- 回溯算法和动态规划到底谁是谁爹

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



扫描线技巧：安排会议室

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

253. 会议室 II（中等）

之前面试，被问到一道非常经典且非常实用的算法题目：会议室安排问题。

力扣上类似的问题是会员题目，你可能没办法做，但对于这种经典的算法题，掌握思路还是必要的。

先说下题目，力扣第 253 题「会议室 II」：

给你输入若干形如 `[begin, end]` 的区间，代表若干会议的开始时间和结束时间，请你计算至少需要申请多少间会议室。

函数签名如下：

```
// 返回需要申请的会议室数量
int minMeetingRooms(int[][] meetings);
```

比如给你输入 `meetings = [[0,30],[5,10],[15,20]]`，算法应该返回 2，因为后两个会议和第一个会议时间是冲突的，至少申请两个会议室才能让所有会议顺利进行。

如果会议之间的时间有重叠，那就得额外申请会议室来开会，想求至少需要多少间会议室，就是让你计算同一时刻最多有多少会议在同时进行。

换句话说，**如果把每个会议的起始时间看做一个线段区间，那么题目就是让你求最多有几个重叠区间**，仅此而已。

对于这种时间安排的问题，本质上讲就是区间调度问题，十有八九得排序，然后找规律来解决。

题目延伸

我们之前写过很多区间调度相关的文章，这里就顺便帮大家梳理一下这类问题的思路：

第一个场景，假设现在只有一个会议室，还有若干会议，你如何将尽可能多的会议安排到这个会议室里？

这个问题需要将这些会议（区间）按结束时间（右端点）排序，然后进行处理，详见前文 贪心算法做时间管理。

第二个场景，给你若干较短的视频片段，和一个较长的视频片段，请你从较短的片段中尽可能少地挑出一些片段，拼接出较长的这个片段。

这个问题需要将这些视频片段（区间）按开始时间（左端点）排序，然后进行处理，详见前文 剪视频剪出一个贪心算法。

第三个场景，给你若干区间，其中可能有些区间比较短，被其他区间完全覆盖住了，请你删除这些被覆盖的区间。

这个问题需要将这些区间按左端点排序，然后就能找到并删除那些被完全覆盖的区间了，详见前文 删除覆盖区间。

第四个场景，给你若干区间，请你将所有有重叠部分的区间进行合并。

这个问题需要将这些区间按左端点排序，方便找出存在重叠的区间，详见前文 合并重叠区间。

第五个场景，有两个部门同时预约了同一个会议室的若干时间段，请你计算会议室的冲突时段。

这个问题就是给你两组区间列表，请你找出这两组区间的交集，这需要你将这些区间按左端点排序，详见前文 区间交集问题。

第六个场景，假设现在只有一个会议室，还有若干会议，如何安排会议才能使这个会议室的闲置时间最少？

这个问题需要动动脑筋，说白了这就是个 0-1 背包问题的变形：

会议室可以看做一个背包，每个会议可以看做一个物品，物品的价值就是会议的时长，请问你如何选择物品（会议）才能最大化背包中的价值（会议室的使用时长）？

当然，这里背包的约束不是一个最大重量，而是各个物品（会议）不能互相冲突。把各个会议按照结束时间进行排序，然后参考前文 0-1 背包问题详解 的思路即可解决，等我以后有机会可以写一写这个问题。

第七个场景，就是本文想讲的场景，给你若干会议，让你合理申请会议室。

好了，举例了这么多，来看看今天的这个问题如何解决。

题目分析

重复一下题目的本质：

给你输入若干时间区间，让你计算同一时刻「最多」有几个区间重叠。

题目的关键点在于，给你任意一个时刻，你是否能够说出这个时刻有几个会议？

如果可以做到，那我遍历所有的时刻，找个最大值，就是需要申请的会议室数量。

有没有一种数据结构或者算法，给我输入若干区间，我能知道每个位置有多少个区间重叠？

老读者肯定可以联想到之前说过的一个算法技巧：差分数组技巧。

把时间线想象成一个初始值为 0 的数组，每个时间区间 `[i, j]` 就相当于一个子数组，这个时间区间有一个会议，那我就把这个子数组中的元素都加一。

最后，每个时刻有几个会议我不就知道了吗？我遍历整个数组，不就知道至少需要几间会议室了吗？

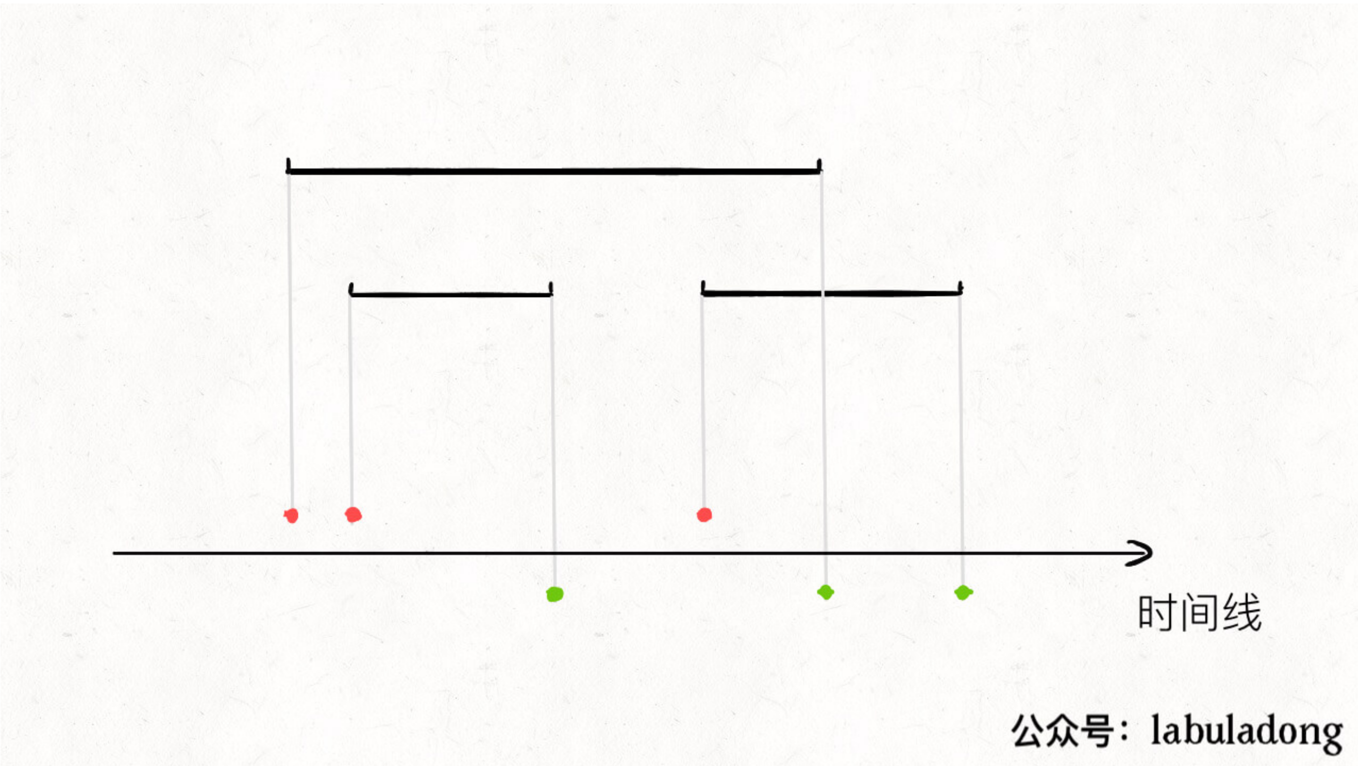
举例来说，如果输入 meetings = [[0,30],[5,10],[15,20]]，那么我们就给数组中 [0,30],[5,10],[15,20] 这几个索引区间分别加一，最后遍历数组，求个最大值就行了。

还记得吗，差分数组技巧可以在 O(1) 时间对整个区间的元素进行加减，所以可以拿来解决这道题。

不过，这个解法的效率不算高，所以我这里不准备具体写差分数组的解法，参照 差分数组技巧 的原理，有兴趣的读者可以自己尝试去实现。

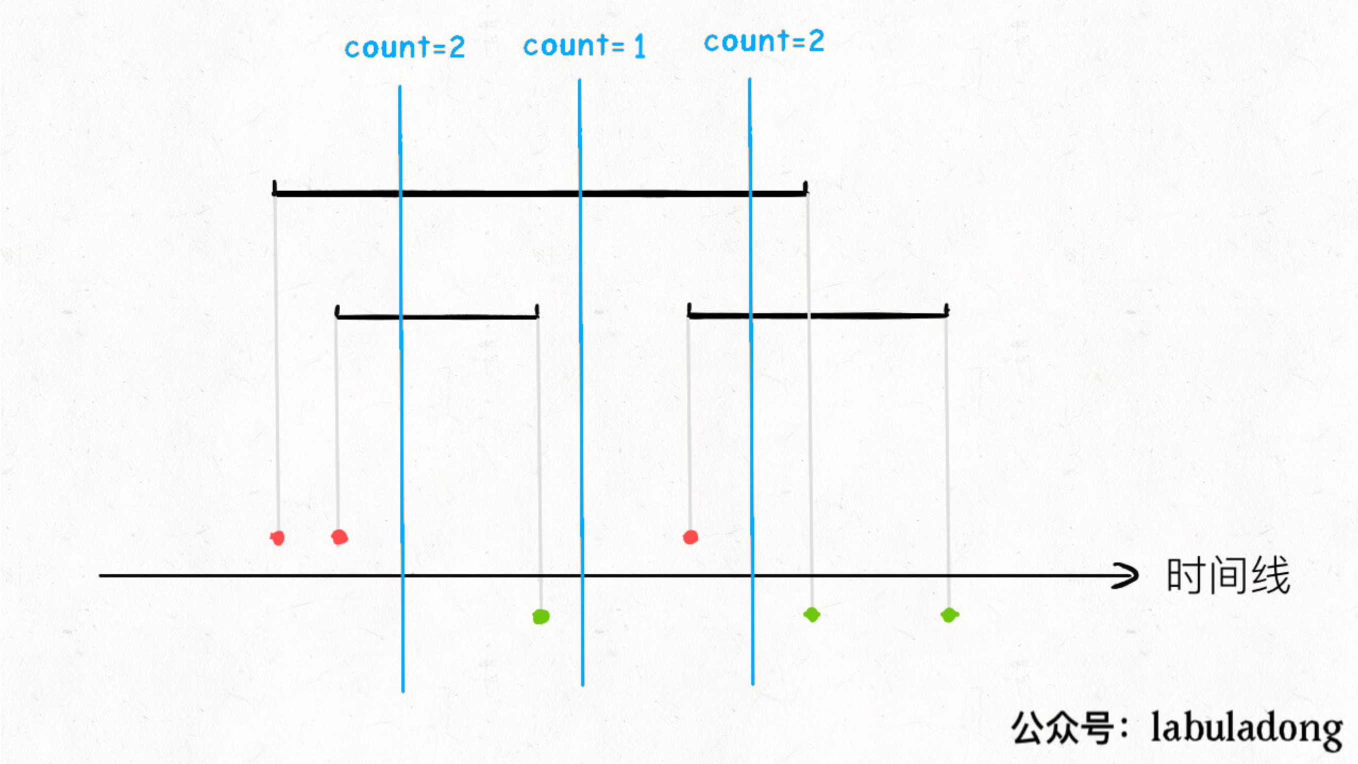
基于差分数组的思路，我们可以推导出一种更高效，更优雅的解法。

我们首先把这些会议的时间区间进行投影：



红色的点代表每个会议的开始时间点，绿色的点代表每个会议的结束时间点。

现在假想有一条带着计数器的线，在时间线上从左至右进行扫描，每遇到红色的点，计数器 count 加一，每遇到绿色的点，计数器 count 减一：



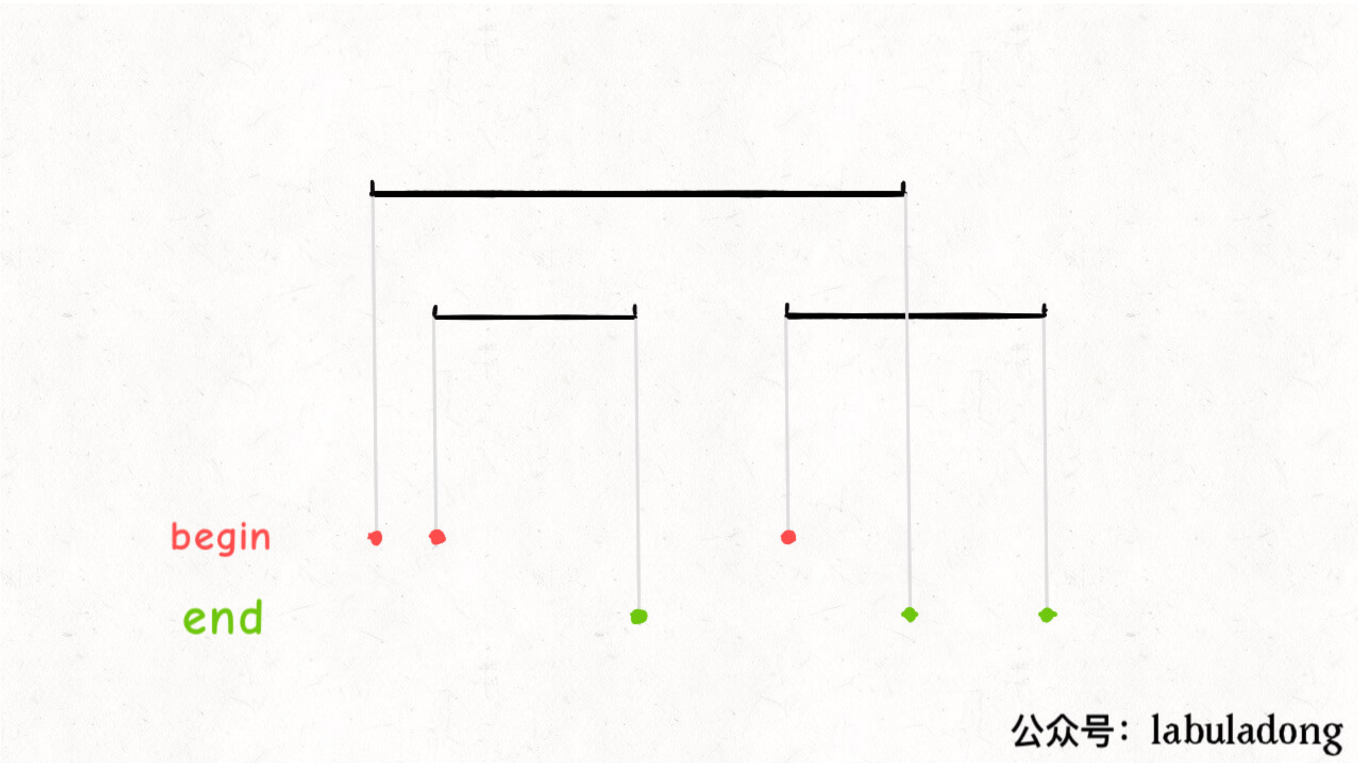
这样一来，每个时刻有多少个会议在同时进行，就是计数器 `count` 的值，`count` 的最大值，就是需要申请的会议室数量。

对差分数组技巧熟悉的读者一眼就能看出来了，这个扫描线其实就是差分数组的遍历过程，所以我们说这是差分数组技巧衍生出来的解法。

代码实现

那么，如何写代码实现这个扫描的过程呢？

首先，对区间进行投影，就相当于对每个区间的起点和终点分别进行排序：



```java
int minMeetingRooms(int[][] meetings) {
    int n = meetings.length;
    int[] begin = new int[n];
    int[] end = new int[n];
    // 把左端点和右端点单独拿出来
    for(int i = 0; i < n; i++) {
        begin[i] = meetings[i][0];
        end[i] = meetings[i][1];
    }
    // 排序后就是图中的红点
    Arrays.sort(begin);
    // 排序后就是图中的绿点
    Arrays.sort(end);

    // ...
}
```

然后就简单了，扫描线从左向右前进，遇到红点就对计数器加一，遇到绿点就对计数器减一，计数器 `count` 的最大值就是答案：

```java
int minMeetingRooms(int[][] meetings) {
    int n = meetings.length;
    int[] begin = new int[n];
    int[] end = new int[n];
    for(int i = 0; i < n; i++) {
        begin[i] = meetings[i][0];
        end[i] = meetings[i][1];
    }
    Arrays.sort(begin);
    Arrays.sort(end);

    // 扫描过程中的计数器
    int count = 0;
    // 双指针技巧
    int res = 0, i = 0, j = 0;
    while (i < n && j < n) {
        if (begin[i] < end[j]) {
            // 扫描到一个红点
            count++;
            i++;
        } else {
            // 扫描到一个绿点
            count--;
            j++;
        }
        // 记录扫描过程中的最大值
        res = Math.max(res, count);
    }
```

```
        return res;
    }
```

这里使用的是 双指针技巧，根据 `i`, `j` 的相对位置模拟扫描线前进的过程。

至此，这道题就做完了。当然，这个题目也可以变形，比如给你若干会议，问你 `k` 个会议室够不够用，其实你套用本文的解法代码，也可以很轻松解决。

接下来可阅读：

区间问题系列合集

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

————————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



当老司机学会了贪心算法

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

134. 加油站（中等）

今天讲一个贪心的老司机的故事，就是力扣第 134 题「加油站」：

134. 加油站 labuladong 题解 思路

难度 中等 687 收藏 分享 切换为英文 接收动态 反馈

在一条环路上有 *N* 个加油站，其中第 *i* 个加油站有汽油 `gas[i]` 升。

你有一辆油箱容量无限的的汽车，从第 *i* 个加油站开往第 *i*+1 个加油站需要消耗汽油 `cost[i]` 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

- 如果题目有解，该答案即为唯一答案。
- 输入数组均为非空数组，且长度相同。
- 输入数组中的元素均为非负数。

示例 1:

```
输入:
gas  = [1,2,3,4,5]
cost = [3,4,5,1,2]

输出: 3

解释:
从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油
开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油
开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油
开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油
开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油
开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。
因此，3 可为起始索引。
```

题目应该不难理解，就是每到达一个站点 `i`，可以加 `gas[i]` 升油，但离开站点 `i` 需要消耗 `cost[i]` 升油，问你从哪个站点出发，可以兜一圈回来。

要说暴力解法，肯定很容易想到，用一个 for 循环遍历所有站点，假设为起点，然后再套一层 for 循环，判断一下是否能够转一圈回到起点：

```
int n = gas.length;
for (int start = 0; start < n; start++) {
    for (int step = 0; step < n; step++) {
        int i = (start + step) % n;
        tank += gas[i];
        tank -= cost[i];
        // 判断油箱中的油是否耗尽
    }
}
```

很明显时间复杂度是 `O(N^2)`，这么简单粗暴的解法一定不是最优的，我们试图分析一下是否有优化的余地。

暴力解法是否有重复计算的部分？是否可以抽象出「状态」，是否对同一个「状态」重复计算了多次？

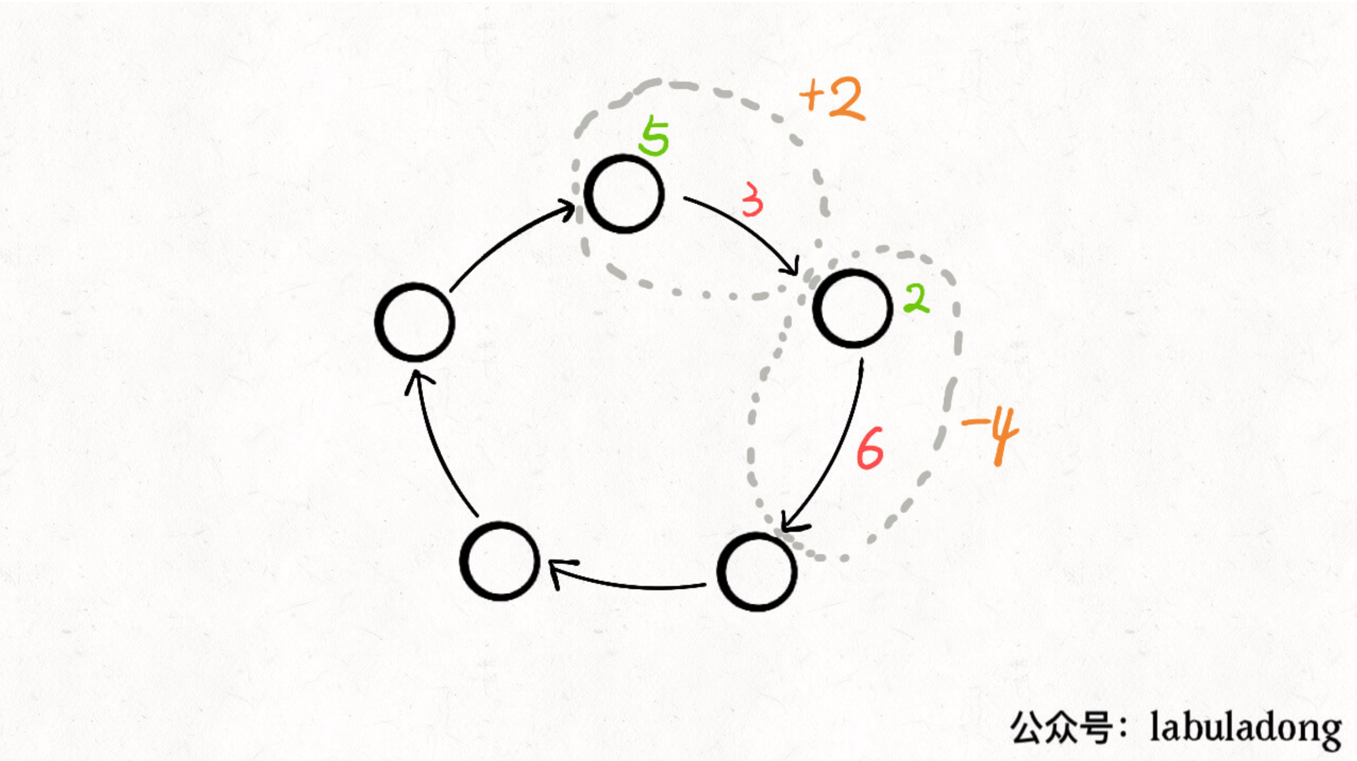
我们前文 动态规划详解 说过，变化的量就是「状态」。那么观察这个暴力穷举的过程，变化的量有两个，分别是「起点」和「当前油箱的油量」，但这两个状态的组合肯定有不下 `O(N^2)` 种，显然没有任何优化的空间。

所以说这道题肯定不是通过简单的剪枝来优化暴力解法的效率，而是需要我们发现一些隐藏较深的规律，从而减少一些冗余的计算。

下面我们介绍两种方法巧解这道题，分别是数学图像解法和贪心解法。

图像解法

汽车进入站点 `i` 可以加 `gas[i]` 的油，离开站点会损耗 `cost[i]` 的油，那么可以把站点和与其相连的路看做一个整体，将 `gas[i] - cost[i]` 作为经过站点 `i` 的油量变化值：



这样，题目描述的场景就被抽象成了一个环形数组，数组中的第 `i` 个元素就是 `gas[i] - cost[i]`。

有了这个环形数组，我们需要判断这个环形数组中是否能够找到一个起点 `start`，使得从这个起点开始的累加和一直大于等于 0。

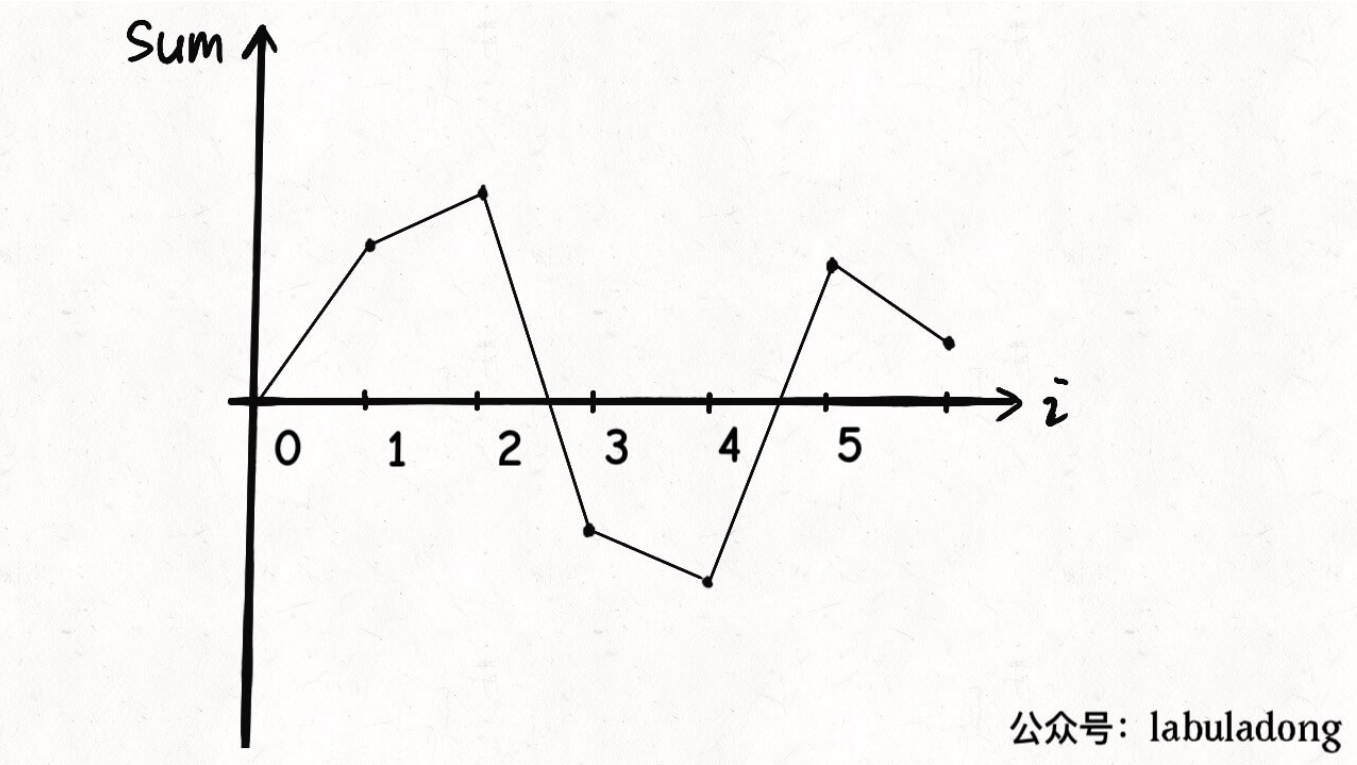
如何判断是否存在这样一个起点 `start`？又如何计算这个起点 `start` 的值呢？

我们不妨就把 0 作为起点，计算累加和的代码非常简单：

```
int n = gas.length, sum = 0;
for (int i = 0; i < n; i++) {
    // 计算累加和
```

```
        sum += gas[i] - cost[i];
    }
```

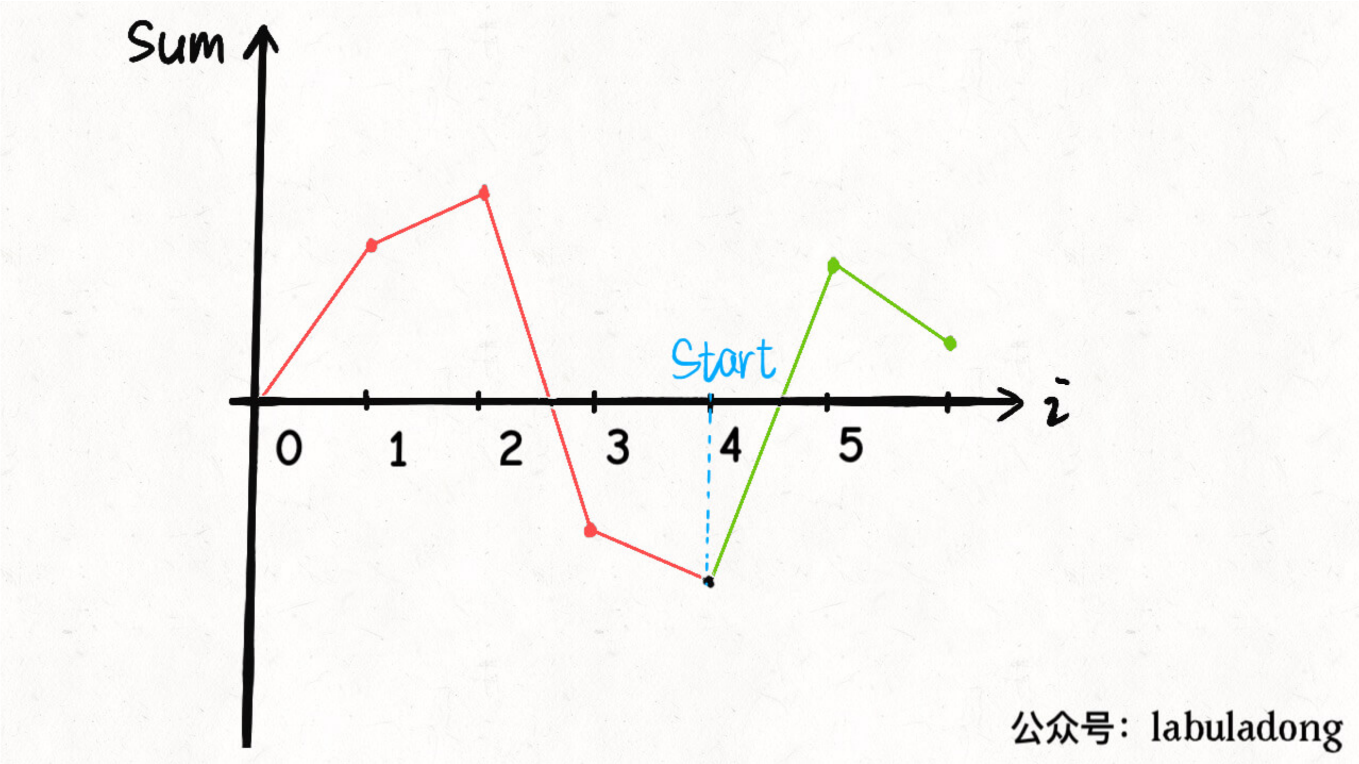
`sum` 就相当于是油箱中油量的变化，上述代码中 `sum` 的变化过程可能是这样的：



显然，上图将 0 作为起点肯定是不行的，因为 `sum` 在变化的过程中小于 0 了，不符合我们「累加和一直大于等于 0」的要求。

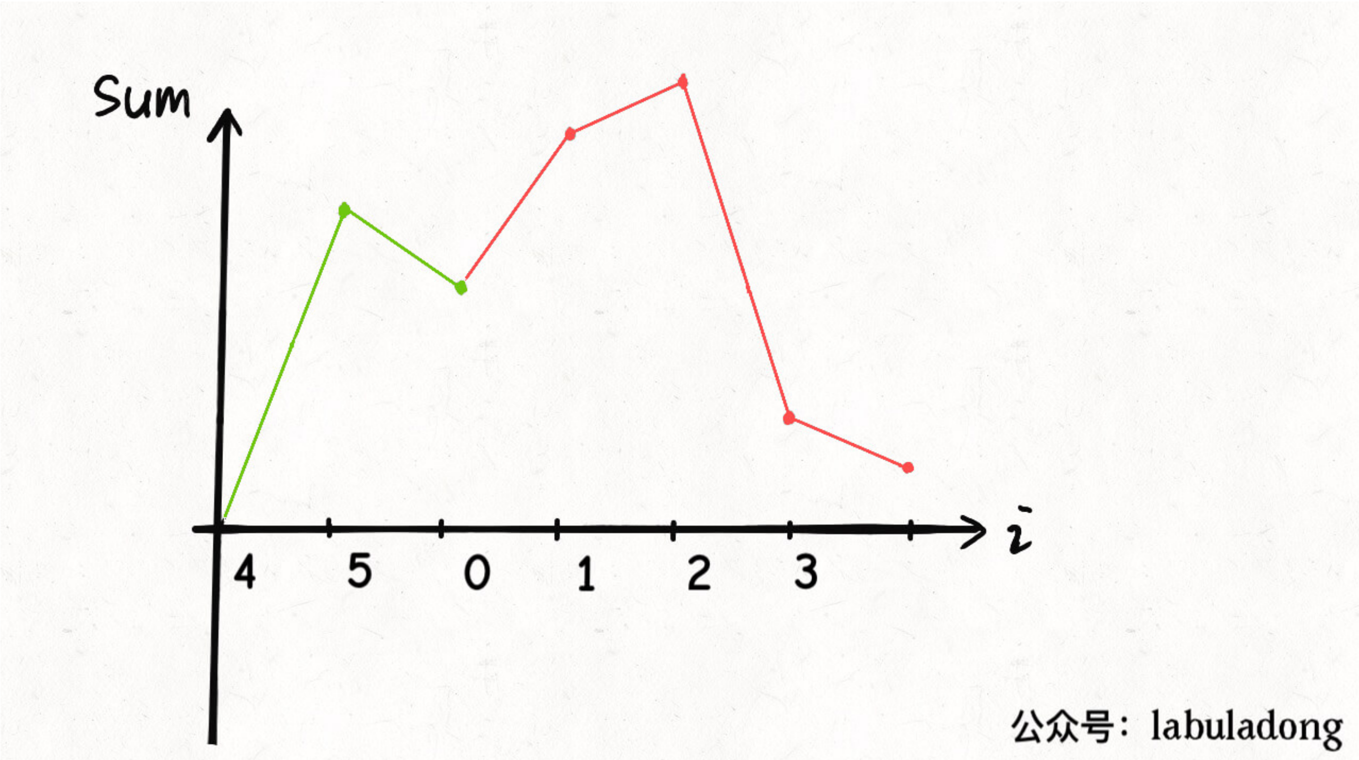
那如果 0 不能作为起点，谁可以作为起点呢？

看图说话，图像的最低点最有可能可以作为起点：



如果把这个「最低点」作为起点，就是说将这个点作为坐标轴原点，就相当于把图像「最大限度」向上平移了。

再加上这个数组是环形数组，最低点左侧的图像可以接到图像的最右侧：



这样，整个图像都保持在 x 轴以上，所以这个最低点 4，就是题目要求我们找的起点。

不过，经过平移后图像一定全部在 x 轴以上吗？不一定，因为还有无解的情况：

如果 `sum(gas[...]) < sum(cost[...])`，总油量小于总的消耗，那肯定是没办法环游所有站点的。

综上，我们就可以写出代码：

```
int canCompleteCircuit(int[] gas, int[] cost) {
    int n = gas.length;
    // 相当于图像中的坐标点和最低点
    int sum = 0, minSum = 0;
    int start = 0;
    for (int i = 0; i < n; i++) {
        sum += gas[i] - cost[i];
        if (sum < minSum) {
            // 经过第 i 个站点后，使 sum 到达新低
            // 所以站点 i + 1 就是最低点（起点）
            start = i + 1;
            minSum = sum;
        }
    }
    if (sum < 0) {
        // 总油量小于总的消耗，无解
        return -1;
    }
    // 环形数组特性
    return start == n ? 0 : start;
}
```

以上是观察函数图像得出的解法，时间复杂度为 O(N)，比暴力解法的效率高很多。

下面我们介绍一种使用贪心思路写出的解法，和上面这个解法比较相似，不过分析过程不尽相同。

贪心解法

用贪心思路解决这道题的关键在于以下这个结论：

如果选择站点 `i` 作为起点「恰好」无法走到站点 `j`，那么 `i` 和 `j` 中间的任意站点 `k` 都不可能作为起点。

比如说，如果从站点 `1` 出发，走到站点 `5` 时油箱中的油量「恰好」减到了负数，那么说明站点 `1` 「恰好」无法到达站点 `5`；那么你从站点 `2,3,4` 任意一个站点出发都无法到达 `5`，因为到达站点 `5` 时油箱的油量也必然被减到负数。

如何证明这个结论？

假设 `tank` 记录当前油箱中的油量，如果从站点 `i` 出发（`tank = 0`），走到 `j` 时恰好出现 `tank < 0` 的情况，那说明走到 `i, j` 之间的任意站点 `k` 时都满足 `tank > 0`，对吧。

如果把 `k` 作为起点的话，相当于在站点 `k` 时 `tank = 0`，那走到 `j` 时必然有 `tank < 0`，也就是说 `k` 肯定不能是起点。

拜托，从 `i` 出发走到 `k` 好歹 `tank > 0`，都无法达到 `j`，现在你还让 `tank = 0` 了，那更不可能走到 `j` 了对吧。

综上，这个结论就被证明了。

回想一下我们开头说的暴力解法是怎么做的？

如果我发现从 i 出发无法走到 j，那么显然 i 不可能是起点。

现在，我们发现了一个新规律，可以推导出什么？

如果我发现从 i 出发无法走到 j，那么 i 以及 i, j 之间的所有站点都不可能作为起点。

看到冗余计算了吗？看到优化的点了吗？

这就是贪心思路的本质，如果找不到重复计算，那就通过问题中一些隐藏较深的规律，来减少冗余计算。

根据这个结论，就可以写出如下代码：

```
int canCompleteCircuit(int[] gas, int[] cost) {
    int n = gas.length;
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += gas[i] - cost[i];
    }
    if (sum < 0) {
        // 总油量小于总的消耗，无解
        return -1;
    }
    // 记录油箱中的油量
    int tank = 0;
    // 记录起点
    int start = 0;
    for (int i = 0; i < n; i++) {
        tank += gas[i] - cost[i];
        if (tank < 0) {
            // 无法从 start 走到 i
            // 所以站点 i + 1 应该是起点
            tank = 0;
            start = i + 1;
        }
    }
    return start == n ? 0 : start;
}
```

这个解法的时间复杂度也是 O(N)，和之前图像法的解题思路有所不同，但代码非常类似。

其实，你可以把这个解法的思路结合图像来思考，可以发现它们本质上是一样的，只是理解方式不同而已。

对于这种贪心算法，没有特别套路化的思维框架，主要还是靠多做题多思考，将题目的场景进行抽象的联想，找出隐藏其中的规律，从而减少计算量，进行效率优化。

好了，这道题就讲到这里，希望对你拓宽思路有帮助。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜

labuladong公众号

剪视频剪出一个贪心算法

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 [点这里](#)。

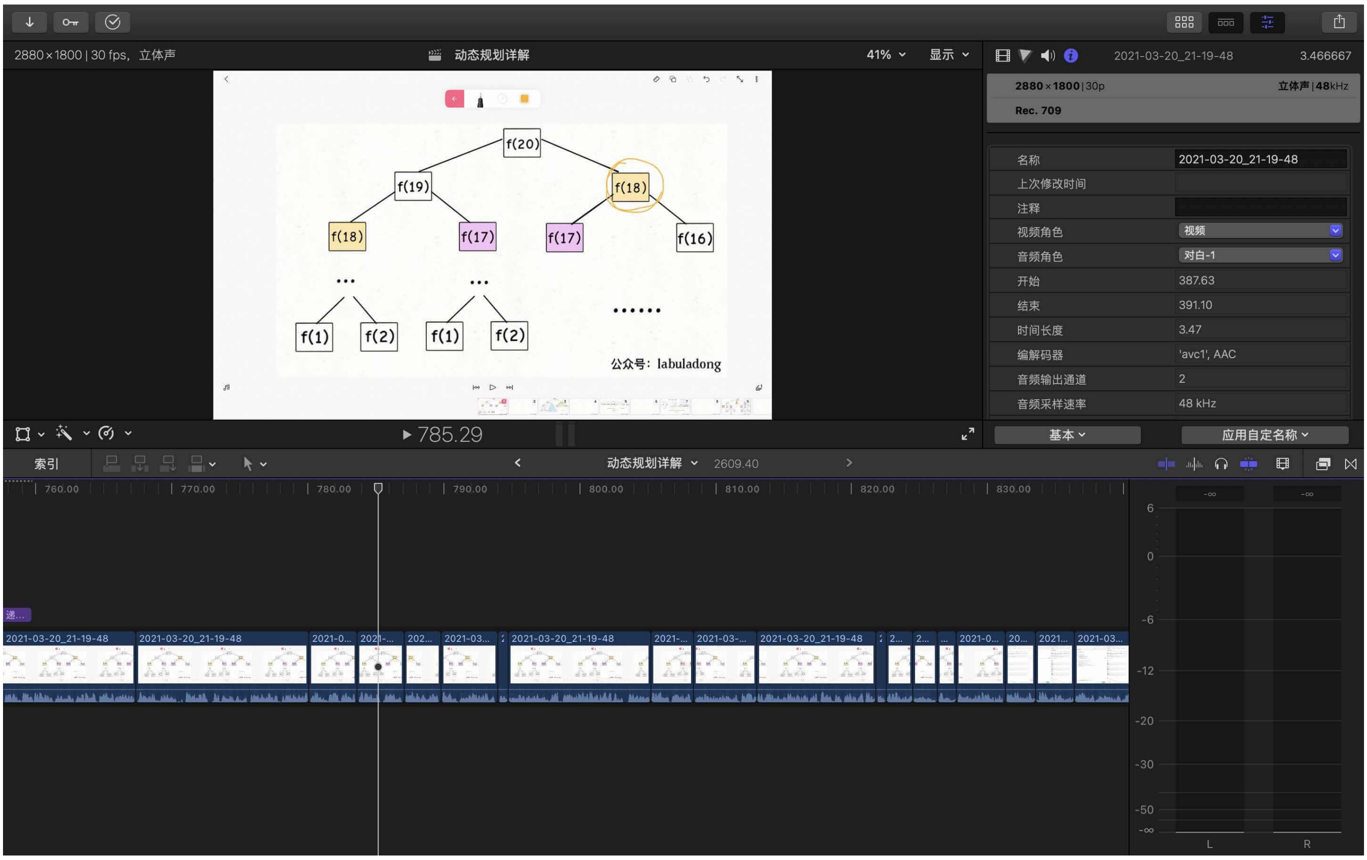
读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

[1024. 视频拼接（中等）](#)

前面发过几个视频，也算是对视频剪辑入了个门。像我这种非专业剪辑玩家，不做什么宏大特效电影镜头，只是做个视频教程，其实也没啥难度，只需要把视频剪流畅，所以用到最多的功能就是切割功能，然后删除和拼接视频片接。

没有剪过视频的读者可能不知道，在常用的剪辑软件中视频被切割成若干片段之后，每个片段都可以还原成原始视频。

就比如一个 10 秒的视频，在中间切一刀剪成两个 5 秒的视频，这两个五秒的视频各自都可以还原成 10 秒的原视频。就好像蚯蚓，把自己切成 4 段就能搓麻，把自己切成 11 段就可以凑一个足球队。



剪视频时，每个视频片段都可以抽象成了一个个区间，时间就是区间的端点，这些区间有的相交，有的不相交……

假设剪辑软件不支持将视频片段还原成原视频，那么如果给我若干视频片段，我怎么将它们还原成原视频呢？

这是个很有意思的区间算法问题，也是力扣第 1024 题「视频拼接」，题目如下：

1024. 视频拼接 labuladong 题解 思路

难度 中等 228

你将会获得一系列视频片段，这些片段来自于一项持续时长为 `T` 秒的体育赛事。这些片段可能有所重叠，也可能长度不一。

视频片段 `clips[i]` 都用区间进行表示：开始于 `clips[i][0]` 秒并于 `clips[i][1]` 秒结束。我们甚至可以对这些片段自由地再剪辑，例如片段 `[0, 7]` 可以剪切成 `[0, 1] + [1, 3] + [3, 7]` 三部分。

我们需要将这些片段进行再剪辑，并将剪辑后的内容拼接成覆盖整个运动过程的片段（`[0, T]`）。返回所需片段的最小数目，如果无法完成该任务，则返回 `-1` 。

示例 1：

```
输入：clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], T = 10
输出：3
解释：
[0,2], [8,10], [1,9] 这三个片段可以还原。
首先从 [1,9] 中剪辑出片段 [2,8]，我们手上就有 [0,2] + [2,8] +
[8,10]，涵盖了整场比赛 [0,10]。
```

函数签名如下：

```
int videoStitching(int[][] clips, int T);
```

记得以前写过好几篇区间相关的问题：

区间问题合集 写过求区间交集、区间并集、区间覆盖这几个问题。

贪心算法做时间管理 写过利用贪心算法求不相交的区间。

算上本文的区间剪辑问题，经典的区间问题也就都讲完了。

思路分析

题目并不难理解，给定一个目标区间和若干小区间，如何通过裁剪和组合小区间拼凑出目标区间？最少需要几个小区间？

前文多次说过，区间问题肯定按照区间的起点或者终点进行排序。

因为排序之后更容易找到相邻区间之间的联系，如果是求最值的问题，可以使用贪心算法进行求解。

区间问题特别容易用贪心算法，公众号历史文章除了 贪心算法之区间调度，还有一篇 贪心算法玩跳跃游戏，其实这个跳跃游戏就相当于一个将起点排序的区间问题，你细品，你细品。

至于到底如何排序，这个就要因题而异了，我做这道题的思路是先按照起点升序排序，如果起点相同的话按照终点降序排序。

为什么这样排序呢，主要考虑到这道题的以下两个特点：

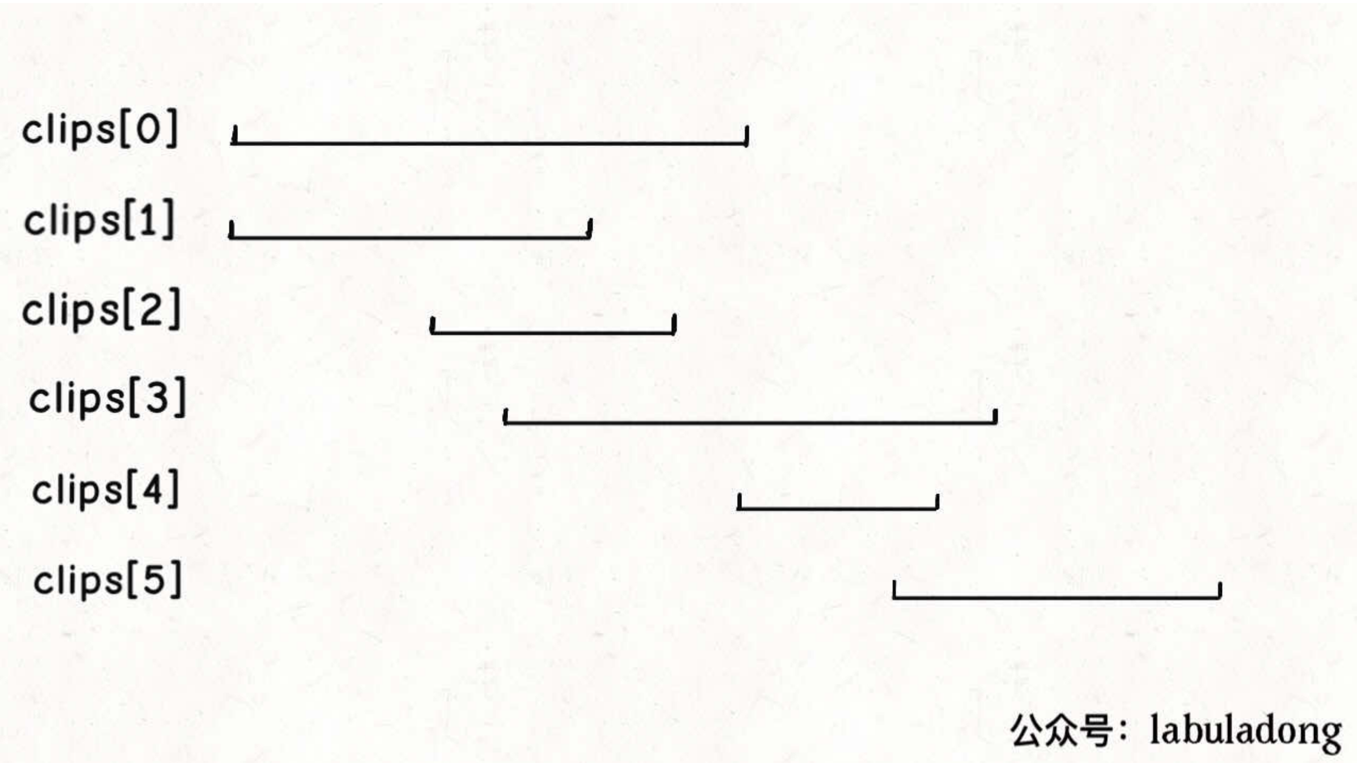
1、要用若干短视频凑出完成视频 `[0, T]`，至少得有一个短视频的起点是 0。

这个很好理解，如果没有一个短视频是从 0 开始的，那么区间 `[0, T]` 肯定是凑不出来的。

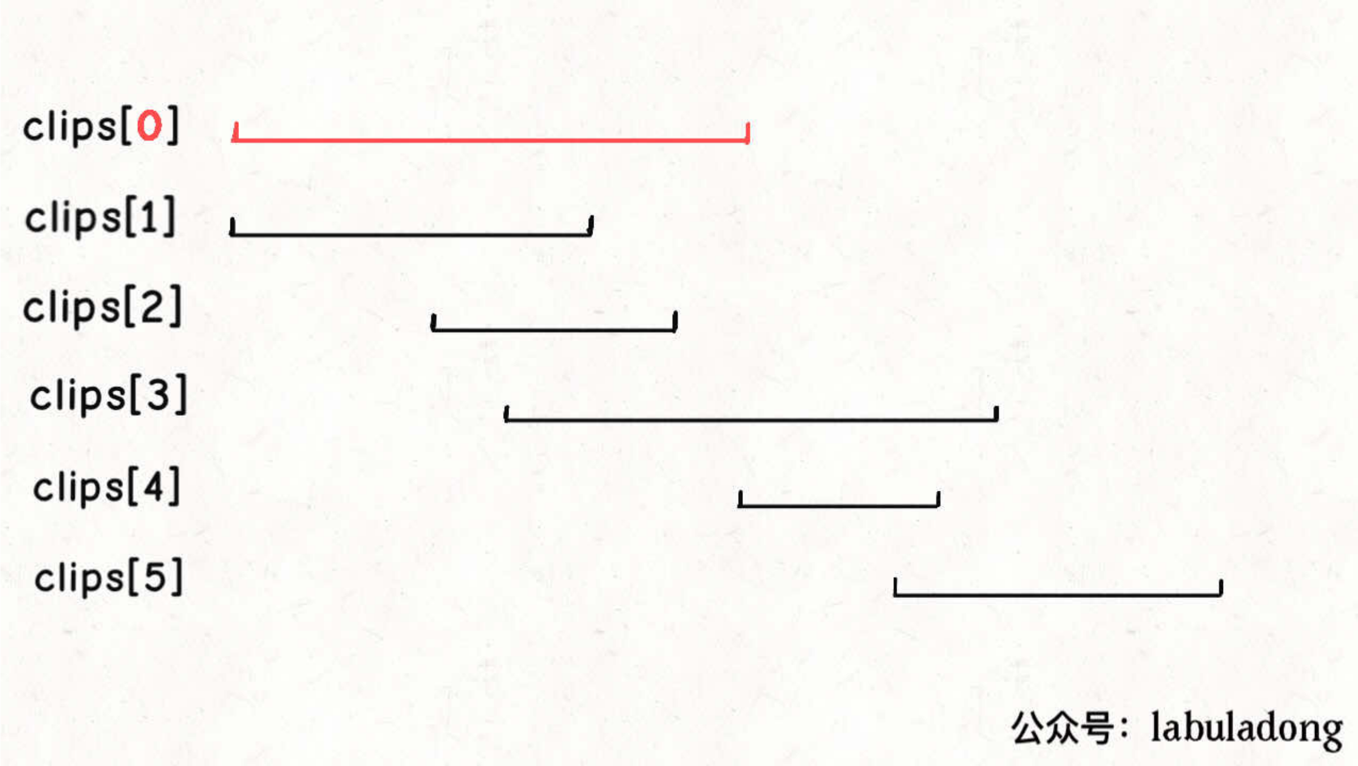
2、如果有几个短视频的起点都相同，那么一定应该选择那个最长（终点最大）的视频。

这一条就是贪心的策略，因为题目让我们计算最少需要的短视频个数，如果起点相同，那肯定是越长越好，不要白不要，多出来了大不了剪辑掉嘛。

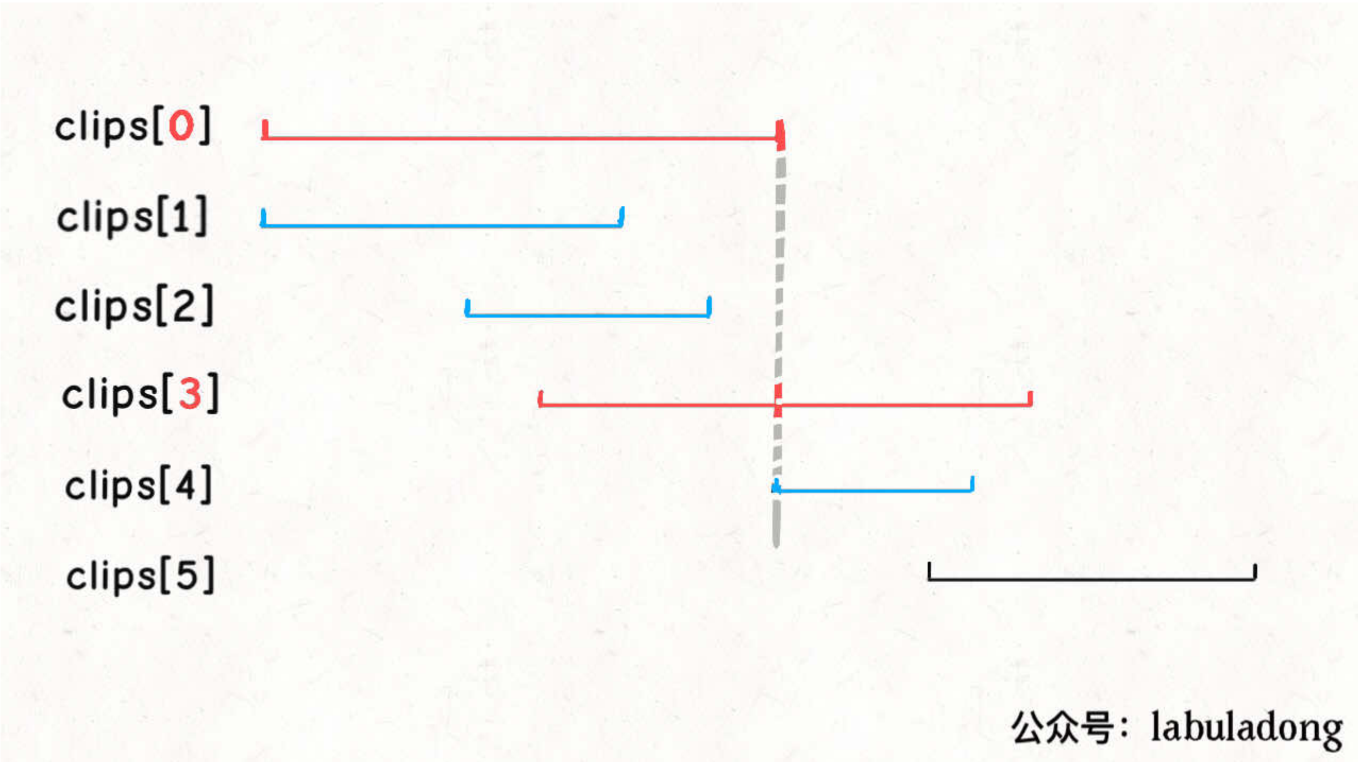
基于以上两个特点，将 `clips` 按照起点升序排序，起点相同的按照终点降序排序，最后得到的区间顺序就像这样：



这样我们就可以确定，如果 `clips[0]` 是的起点是 0，那么 `clips[0]` 这个视频一定会被选择。



当我们确定 `clips[0]` 一定会被选择之后，就可以选出下一个会被选择的视频：



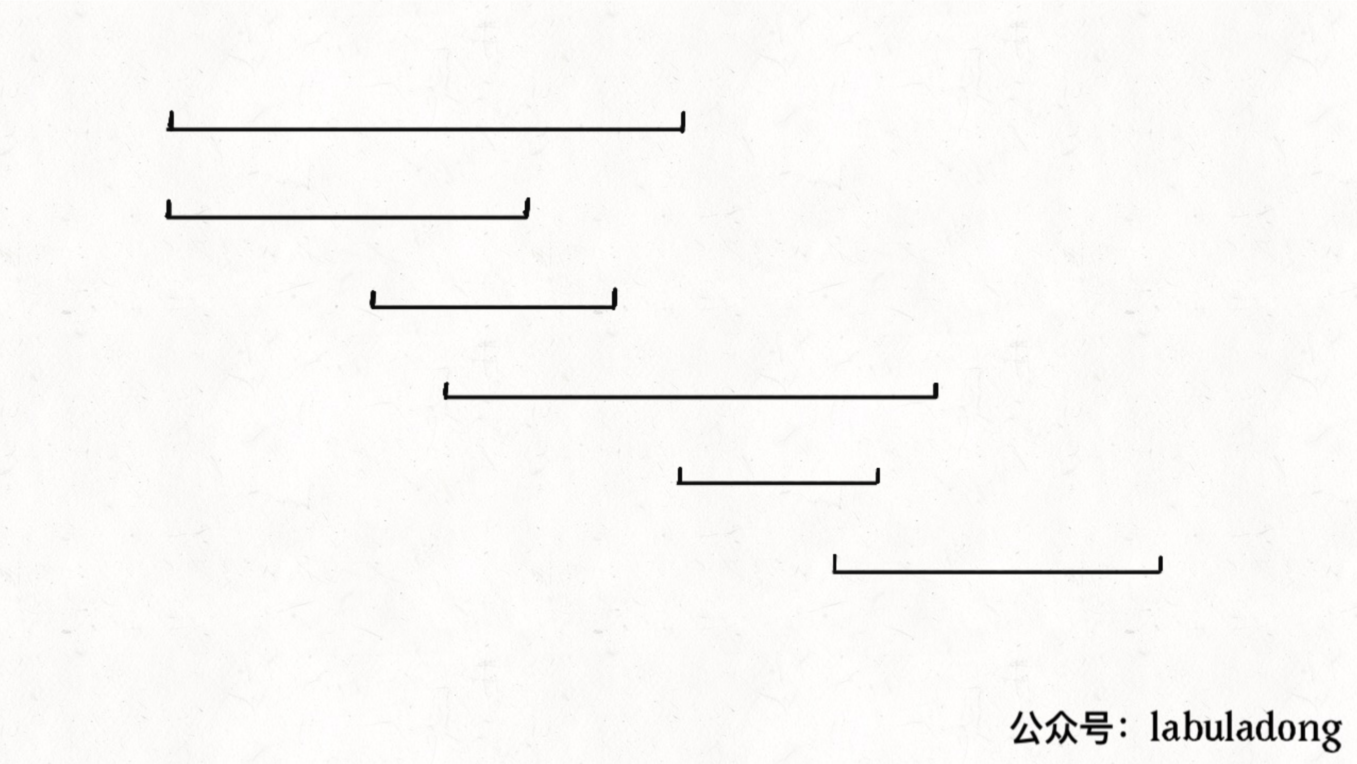
我们会比较所有起点小于 `clips[0][1]` 的区间，根据贪心策略，它们中终点最大的那个区间就是第二个会被选中的视频。

然后可以通过第二个视频区间贪心选择出第三个视频，以此类推，直到覆盖区间 `[0, T]`，或者无法覆盖返回 -1。

以上就是这道题的解题思路，仔细想想，这题的核心和前文 [贪心算法玩跳跃游戏](#) 写的跳跃游戏是相同的，如果你能看出这两者的联系，就可以说理解贪心算法的奥义了。

代码实现

实现上述思路需要我们用两个变量 curEnd 和 nextEnd 来进行：



最终代码实现如下：

```
int videoStitching(int[][] clips, int T) {
    if (T == 0) return 0;
    // 按起点升序排列，起点相同的降序排列
    Arrays.sort(clips, (a, b) -> {
        if (a[0] == b[0]) {
            return b[1] - a[1];
        }
        return a[0] - b[0];
    });
    // 记录选择的短视频个数
    int res = 0;

    int curEnd = 0, nextEnd = 0;
    int i = 0, n = clips.length;
    while (i < n && clips[i][0] <= curEnd) {
        // 在第 res 个视频的区间内贪心选择下一个视频
        while (i < n && clips[i][0] <= curEnd) {
            nextEnd = Math.max(nextEnd, clips[i][1]);
            i++;
        }
        // 找到下一个视频，更新 curEnd
        res++;
        curEnd = nextEnd;
        if (curEnd >= T) {
            // 已经可以拼出区间 [0, T]
```

```
                return res;
            }
        }
        // 无法连续拼出区间 [0, T]
        return -1;
    }
```

这段代码的时间复杂度是多少呢？虽然代码中有一个嵌套的 while 循环，但这个嵌套 while 循环的时间复杂度是 `O(N)`。因为当 `i` 递增到 `n` 时循环就会结束，所以这段代码只会执行 `O(N)` 次。

但是别忘了我们对 `clips` 数组进行了一次排序，消耗了 `O(NlogN)` 的时间，所以本算法的总时间复杂度是 `O(NlogN)`。

最后说一句，我去 B 站做 up 了，B 站搜索同名账号「labuladong」即可关注！

如果你有高质量的问题，可以 [点击这里](#) 进入本文的讨论区。

____

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：



如何实现一个计算器

Stars 103k 知乎 @labuladong 公众号 @labuladong B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

224. 基本计算器（困难）

227. 基本计算器II（中等）

772. 基本计算器III（困难）

我们最终要实现的计算器功能如下：

1、输入一个字符串，可以包含+ - * /、数字、括号以及空格，你的算法返回运算结果。

2、要符合运算法则，括号的优先级最高，先乘除后加减。

3、除号是整数除法，无论正负都向 0 取整（5/2=2，-5/2=-2）。

4、可以假定输入的算式一定合法，且计算过程不会出现整型溢出，不会出现除数为 0 的意外情况。

比如输入如下字符串，算法会返回 9：

`3 * (2-6 /(3 -7))`

可以看到，这就已经非常接近我们实际生活中使用的计算器了，虽然我们以前肯定都用过计算器，但是如果简单思考一下其算法实现，就会大惊失色：

1、按照常理处理括号，要先计算最内层的括号，然后向外慢慢化简。这个过程我们手算都容易出错，何况写成算法呢！

2、要做到先乘除，后加减，这一点教会小朋友还不算难，但教给计算机恐怕有点困难。

3、要处理空格。我们为了美观，习惯性在数字和运算符之间打个空格，但是计算之中得想办法忽略这些空格。

我记得很多大学数据结构的教材上，在讲栈这种数据结构的时候，应该都会用计算器举例，但是有一说一，讲的真的垃圾，不知道多少未来的计算机科学家就被这种简单的数据结构劝退了。

那么本文就来聊聊怎么实现上述一个功能完备的计算器功能，**关键在于层层拆解问题，化整为零，逐个击破**，相信这种思维方式能帮大家解决各种复杂问题。

下面就来拆解，从最简单的一个问题开始。

一、字符串转整数

是的，就是这么一个简单的问题，首先告诉我，怎么把一个字符串形式的**正**整数，转化成 int 型？

```
string s = "458";

int n = 0;
for (int i = 0; i < s.size(); i++) {
    char c = s[i];
    n = 10 * n + (c - '0');
}
// n 现在就等于 458
```

这个还是很简单的吧，老套路了。但是即便这么简单，依然有坑：**`(c - '0')`的这个括号不能省略，否则可能造成整型溢出**。

因为变量`c`是一个 ASCII 码，如果不加括号就会先加后减，想象一下`s`如果接近 INT_MAX，就会溢出。所以用括号保证先减后加才行。

二、处理加减法

现在进一步，**如果输入的这个算式只包含加减法，而且不存在空格**，你怎么计算结果？我们拿字符串算式`1-12+3`为例，来说一个很简单的思路：

1、先给第一个数字加一个默认符号`+`，变成`+1-12+3`。

2、把一个运算符和数字组合成一对儿，也就是三对儿`+1`，`-12`，`+3`，把它们转化成数字，然后放到一个栈中。

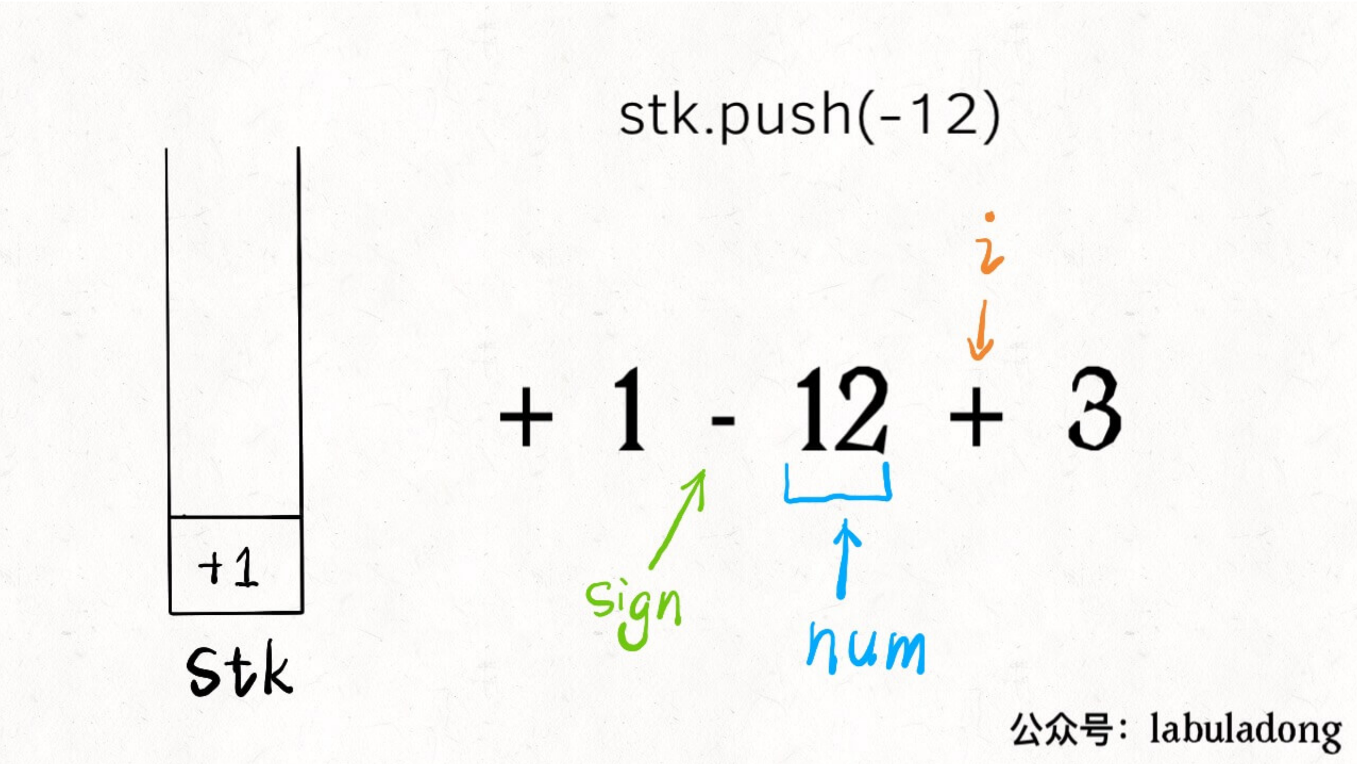
3、将栈中所有的数字求和，就是原算式的结果。

我们直接看代码，结合一张图就看明白了：

```
int calculate(string s) {
    stack<int> stk;
    // 记录算式中的数字
    int num = 0;
    // 记录 num 前的符号，初始化为 +
    char sign = '+';
    for (int i = 0; i < s.size(); i++) {
        char c = s[i];
        // 如果是数字，连续读取到 num
        if (isdigit(c))
            num = 10 * num + (c - '0');
        // 如果不是数字，就是遇到了下一个符号，
        // 之前的数字和符号就要存进栈中
        if (!isdigit(c) || i == s.size() - 1) {
            switch (sign) {
```

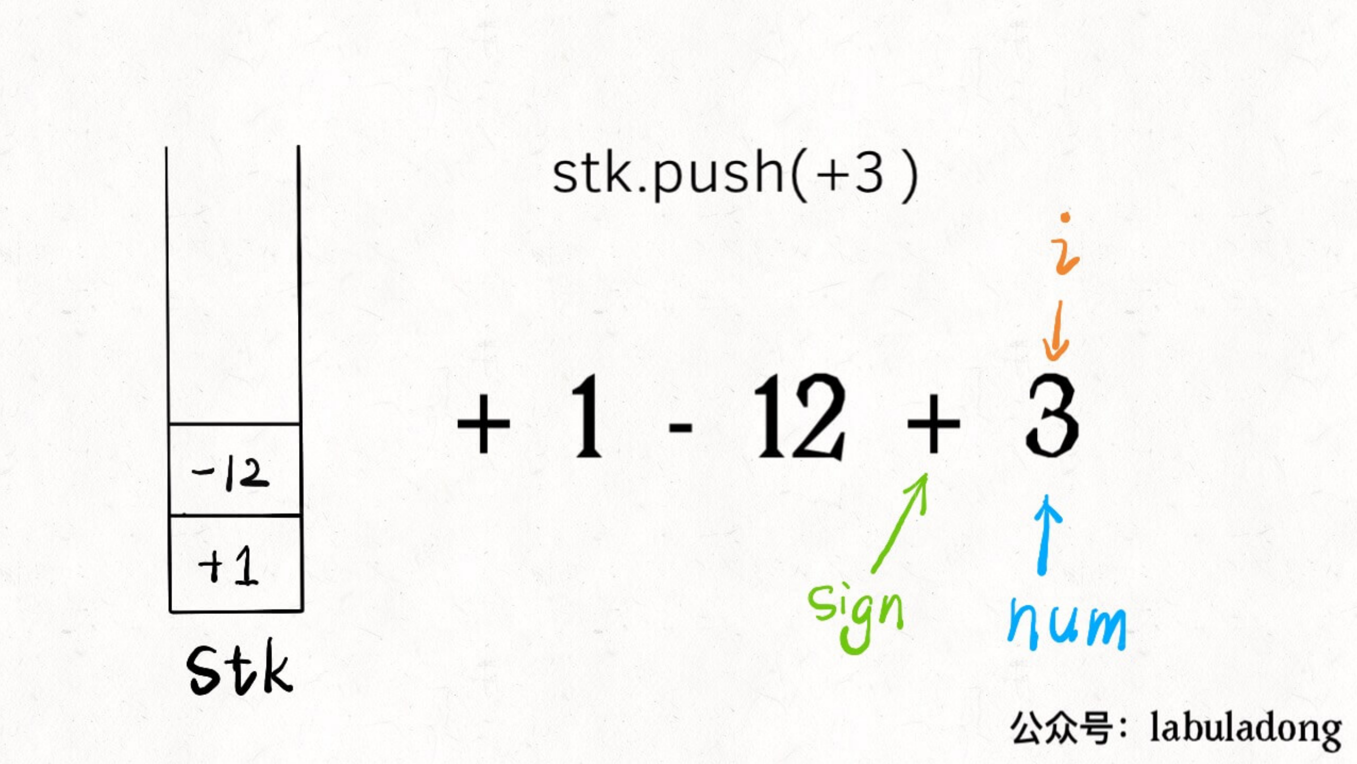
```
                case '+':
                    stk.push(num); break;
                case '-':
                    stk.push(-num); break;
            }
            // 更新符号为当前符号，数字清零
            sign = c;
            num = 0;
        }
    }
    // 将栈中所有结果求和就是答案
    int res = 0;
    while (!stk.empty()) {
        res += stk.top();
        stk.pop();
    }
    return res;
}
```

我估计就是中间带`switch`语句的部分有点不好理解吧，`i`就是从左到右扫描，`sign`和`num`跟在它身后。当`s[i]`遇到一个运算符时，情况是这样的：



所以说，此时要根据`sign`的 case 不同选择`nums`的正负号，存入栈中，然后更新`sign`并清零`nums`记录下一对儿符合和数字的组合。

另外注意，不只是遇到新的符号会触发入栈，当`i`走到了算式的尽头（`i == s.size() - 1`），也应该将前面的数字入栈，方便后续计算最终结果。



至此，仅处理紧凑加减法字符串的算法就完成了，请确保理解以上内容，后续的内容就基于这个框架修修改改就完事儿了。

三、处理乘除法

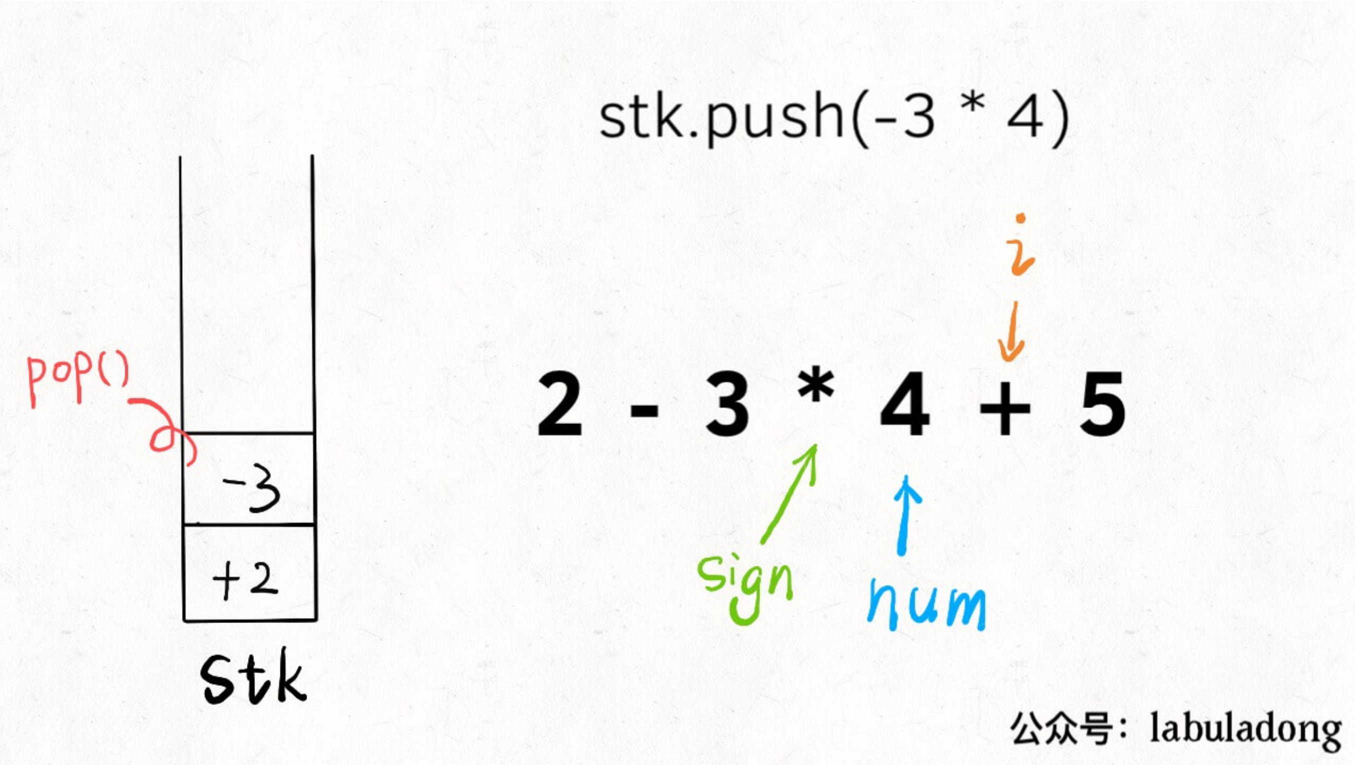
其实思路跟仅处理加减法没啥区别，拿字符串`2-3*4+5`举例，核心思路依然是把字符串分解成符号和数字的组合。

比如上述例子就可以分解为`+2`，`-3`，`*4`，`+5`几对儿，我们刚才不是没有处理乘除号吗，很简单，**其他部分都不用变**，在`switch`部分加上对应的 case 就行了：

```
for (int i = 0; i < s.size(); i++) {
    char c = s[i];
    if (isdigit(c))
        num = 10 * num + (c - '0');

    if (!isdigit(c) || i == s.size() - 1) {
        switch (sign) {
            int pre;
            case '+':
                stk.push(num); break;
            case '-':
                stk.push(-num); break;
            // 只要拿出前一个数字做对应运算即可
            case '*':
                pre = stk.top();
                stk.pop();
                stk.push(pre * num);
                break;
            case '/':
                pre = stk.top();
```

```
                stk.pop();
                stk.push(pre / num);
                break;
        }
        // 更新符号为当前符号，数字清零
        sign = c;
        num = 0;
    }
}
```



乘除法优先于加减法体现在，乘除法可以和栈顶的数结合，而加减法只能把自己放入栈。

现在我们思考一下**如何处理字符串中可能出现的空格字符**。其实也非常简单，想想空格字符的出现，会影响我们现有代码的哪一部分？

```
// 如果 c 非数字
if (!isdigit(c) || i == s.size() - 1) {
    switch (c) {...}
    sign = c;
    num = 0;
}
```

显然空格会进入这个 if 语句，但是我们并不想让空格的情况进入这个 if，因为这里会更新`sign`并清零`nums`，空格根本就不是运算符，应该被忽略。

那么只要多加一个条件即可：

```
if ((!isdigit(c) && c != ' ') || i == s.size() - 1) {
    ...
}
```

好了，现在我们的算法已经可以按照正确的法则计算加减乘除，并且自动忽略空格符，剩下的就是如何让算法正确识别括号了。

四、处理括号

处理算式中的括号看起来应该是最难的，但真没有看起来那么难。

为了规避编程语言的繁琐细节，我把前面解法的代码翻译成 Python 版本：

```python
def calculate(s: str) -> int:

    def helper(s: List) -> int:
        stack = []
        sign = '+'
        num = 0

        while len(s) > 0:
            c = s.pop(0)
            if c.isdigit():
                num = 10 * num + int(c)

            if (not c.isdigit() and c != ' ') or len(s) == 0:
                if sign == '+':
                    stack.append(num)
                elif sign == '-':
                    stack.append(-num)
                elif sign == '*':
                    stack[-1] = stack[-1] * num
                elif sign == '/':
                    # python 除法向 0 取整的写法
                    stack[-1] = int(stack[-1] / float(num))
                num = 0
                sign = c

        return sum(stack)
    # 需要把字符串转成列表方便操作
    return helper(list(s))
```

这段代码跟刚才 C++ 代码完全相同，唯一的区别是，不是从左到右遍历字符串，而是不断从左边`pop`出字符，本质还是一样的。

那么，为什么说处理括号没有看起来那么难呢，**因为括号具有递归性质**。我们拿字符串`3*(4-5/2)-6`举例：

calculate(`3*(4-5/2)-6`) = 3 * calculate(`4-5/2`) - 6 = 3 * 2 - 6 = 0

可以脑补一下，无论多少层括号嵌套，通过 calculate 函数递归调用自己，都可以将括号中的算式化简成一个数字。**换句话说，括号包含的算式，我们直接视为一个数字就行了。**

现在的问题是，递归的开始条件和结束条件是什么？**遇到`(`开始递归，遇到`)`结束递归**：

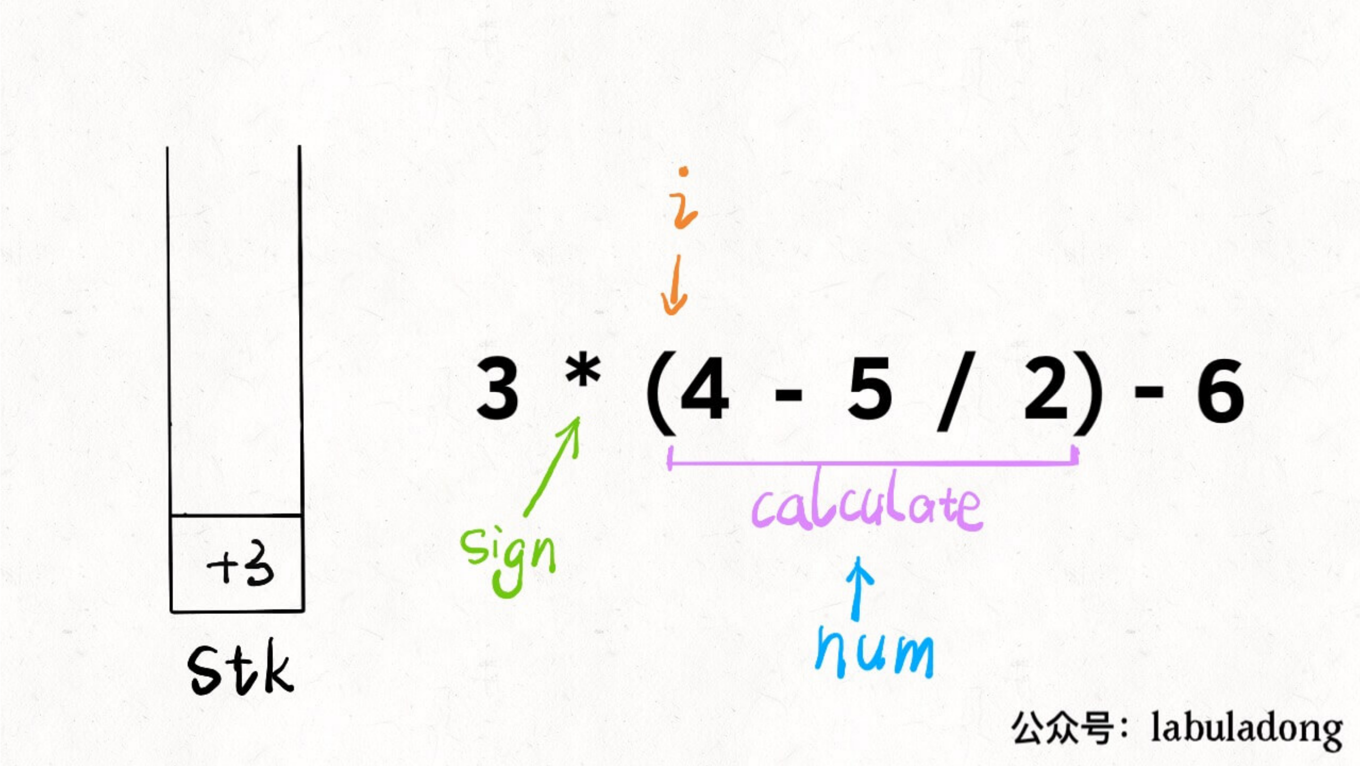
```
def calculate(s: str) -> int:

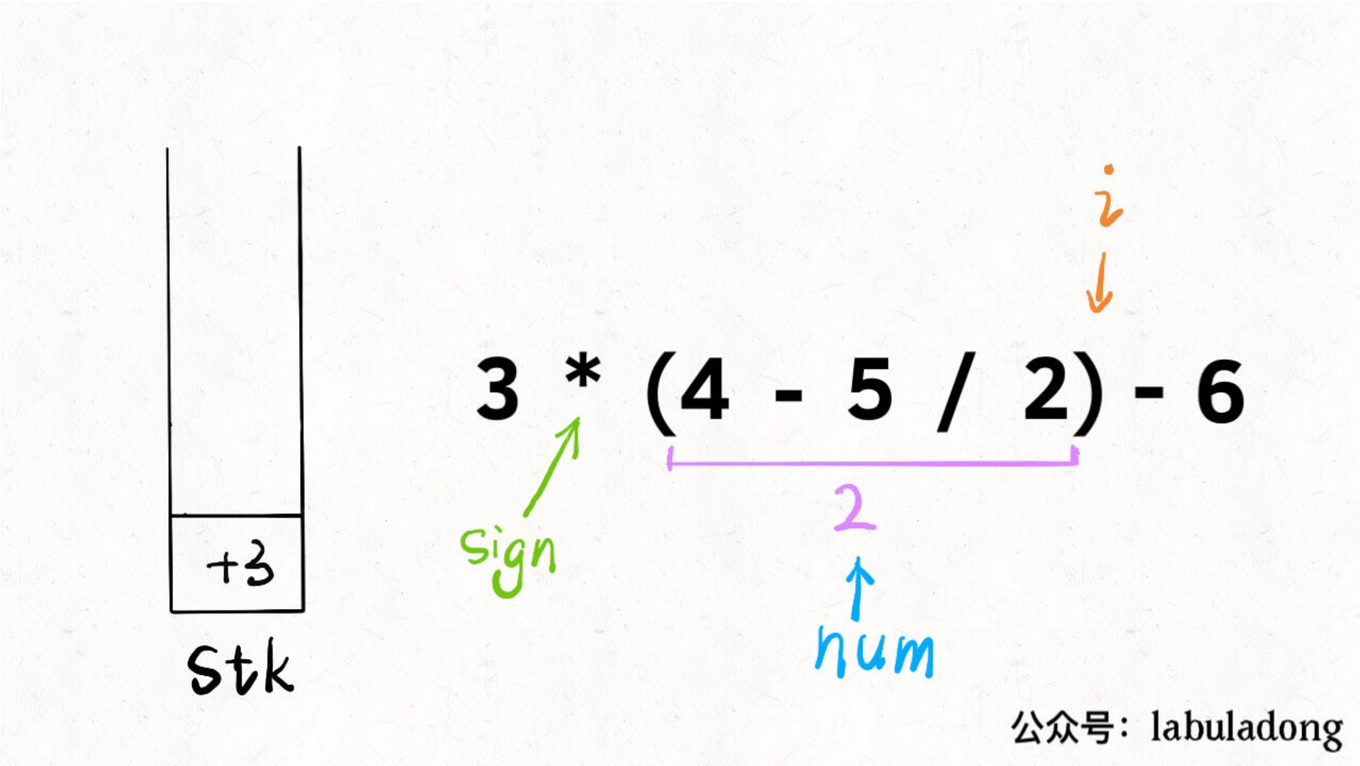
    def helper(s: List) -> int:
        stack = []
        sign = '+'
        num = 0

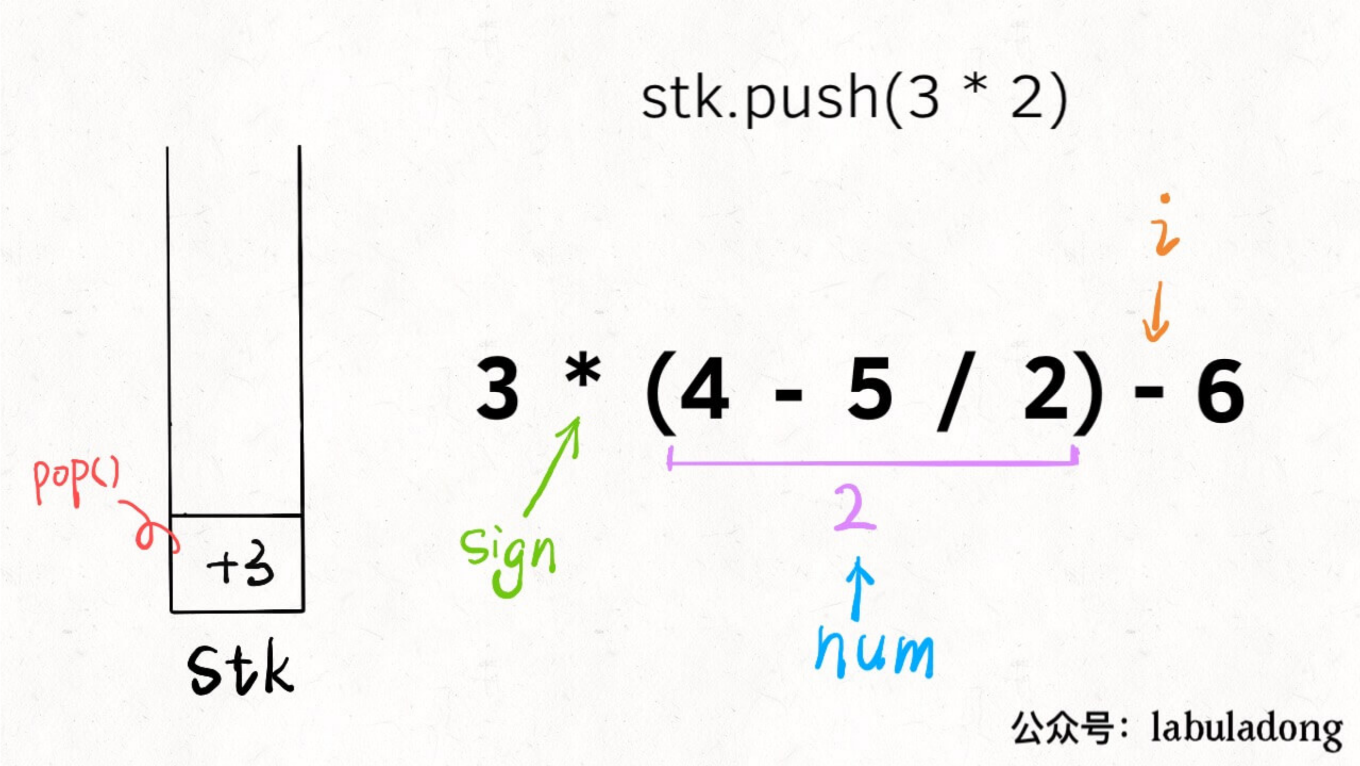
        while len(s) > 0:
            c = s.popleft()
            if c.isdigit():
                num = 10 * num + int(c)
            # 遇到左括号开始递归计算 num
            if c == '(':
                num = helper(s)

            if (not c.isdigit() and c != ' ') or len(s) == 0:
                if sign == '+': ...
                elif sign == '-': ...
                elif sign == '*': ...
                elif sign == '/': ...
                num = 0
                sign = c
            # 遇到右括号返回递归结果
            if c == ')': break
        return sum(stack)

    return helper(collections.deque(s))
```


i
3 * (4 - 5 / 2) - 6
sign
calculate
num
+3
Stk
公众号：labuladong


i
3 * (4 - 5 / 2) - 6
sign
2
num
+3
Stk
公众号：labuladong



你看，加了两三行代码，就可以处理括号了，这就是递归的魅力。至此，计算器的全部功能就实现了，通过对问题的层层拆解化整为零，再回头看，这个问题似乎也没那么复杂嘛。

五、最后总结

本文借实现计算器的问题，主要想表达的是一种处理复杂问题的思路。

我们首先从字符串转数字这个简单问题开始，进而处理只包含加减法的算式，进而处理包含加减乘除四则运算的算式，进而处理空格字符，进而处理包含括号的算式。

可见，对于一些比较困难的问题，其解法并不是一蹴而就的，而是步步推进，螺旋上升的。如果一开始给你原题，你不会做，甚至看不懂答案，都很正常，关键在于我们自己如何简化问题，如何以退为进。

退而求其次是一种很聪明策略。你想想啊，假设这是一道考试题，你不会实现这个计算器，但是你写了字符串转整数的算法并指出了容易溢出的陷阱，那起码可以得 20 分吧；如果你能够处理加减法，那可以得 40 分吧；如果你能处理加减乘除四则运算，那起码够 70 分了；再加上处理空格字符，80 有了吧。我就是不会处理括号，那就算了，80 已经很 OK 了好不好。

如果你有高质量的问题，可以 [点击这里](#) 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

谁能想到，斗地主也能玩出算法

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 点这里。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

659. 分割数组为连续子序列（中等）

斗地主中，大小连续的牌可以作为顺子，有时候我们把对子拆掉，结合单牌，可以组合出更多的顺子，可能更容易赢。

那么如何合理拆分手上的牌，合理地拆出顺子呢？我们今天看一道非常有意思的算法题，连续子序列的划分问题。

这是力扣第 659 题「分割数组为连续子序列」，题目很简单：

给你输入一个**升序排列**的数组 `nums`（可能包含重复数字），请你判断 `nums` 是否能够被分割成若干个**长度至少为 3** 的子序列，每个子序列都由连续的整数组成。

函数签名如下：

```
bool isPossible(vector<int>& nums);
```

比如题目举的例子，输入 `nums = [1,2,3,3,4,4,5,5]`，算法返回 true。

因为 `nums` 可以被分割成 `[1,2,3,4,5]` 和 `[3,4,5]` 两个包含连续整数子序列。

但如果输入 `nums = [1,2,3,4,4,5]`，算法返回 false，因为无法分割成两个长度至少为 3 的连续子序列。

对于这种涉及连续整数的问题，应该条件反射地想到排序，不过题目说了，输入的 `nums` 本就是排好序的。

那么，我们如何判断 `nums` 是否能够被划分成若干符合条件的子序列呢？

类似前文 回溯算法进行集合划分，我们想把 `nums` 的元素划分到若干个子序列中，其实就是下面这个代码逻辑：

```
for (int v : nums) {
    if (...) {
        // 将 v 分配到某个子序列中
```

```
        } else {
            // 实在无法分配 v
            return false;
        }
        return true;
    }
```

关键在于，我们怎么知道当前元素 v 如何进行分配呢？

肯定得分情况讨论，把情况讨论清楚了，题目也就做出来了。

总共有两种情况：

1、当前元素 v 自成一派，「以自己开头」构成一个长度至少为 3 的序列。

比如输入 nums = [1,2,3,6,7,8]，遍历到元素 6 时，它只能自己开头形成一个符合条件的子序列 [6,7,8]。

2、当前元素 v 接到已经存在的子序列后面。

比如输入 nums = [1,2,3,4,5]，遍历到元素 4 时，它只能接到已经存在的子序列 [1,2,3] 后面。它没办法自成开头形成新的子序列，因为少了个 6。

但是，如果这两种情况都可以，应该如何选择？

比如说，输入 nums = [1,2,3,4,5,5,6,7]，对于元素 4，你说它应该形成一个新的子序列 [4,5,6] 还是接到子序列 [1,2,3] 后面呢？

显然，nums 数组的正确划分方法是分成 [1,2,3,4,5] 和 [5,6,7]，所以元素 4 应该**优先判断自己是否能够接到其他序列后面**，如果不行，再判断是否可以作为新的子序列开头。

这就是整体的思路，想让算法代码实现这两个选择，需要两个哈希表来做辅助：

freq 哈希表帮助一个元素判断自己是否能够作为开头，need 哈希表帮助一个元素判断自己是否可以被接到其他序列后面。

freq 记录每个元素出现的次数，比如 freq[3] == 2 说明元素 3 在 nums 中出现了 2 次。

那么如果我发现 freq[3], freq[4], freq[5] 都是大于 0 的，那就说明元素 3 可以作为开头组成一个长度为 3 的子序列。

need 记录哪些元素可以被接到其他子序列后面。

比如说现在已经组成了两个子序列 [1,2,3,4] 和 [2,3,4]，那么 need[5] 的值就应该是 2，说明对元素 5 的需求为 2。

明白了这两个哈希表的作用，我们就可以看懂解法了：

```
bool isPossible(vector<int>& nums) {

    unordered_map<int, int> freq, need;

```

```
    // 统计 nums 中元素的频率
    for (int v : nums) freq[v]++;

    for (int v : nums) {
        if (freq[v] == 0) {
            // 已经被用到其他子序列中
            continue;
        }
        // 先判断 v 是否能接到其他子序列后面
        if (need.count(v) && need[v] > 0) {
            // v 可以接到之前的某个序列后面
            freq[v]--;
            // 对 v 的需求减一
            need[v]--;
            // 对 v + 1 的需求加一
            need[v + 1]++;
        } else if (freq[v] > 0 && freq[v + 1] > 0 && freq[v + 2] > 0) {
            // 将 v 作为开头，新建一个长度为 3 的子序列 [v,v+1,v+2]
            freq[v]--;
            freq[v + 1]--;
            freq[v + 2]--;
            // 对 v + 3 的需求加一
            need[v + 3]++;
        } else {
            // 两种情况都不符合，则无法分配
            return false;
        }
    }

    return true;
}
```

至此，这道题就解决了。

那你可能会说，斗地主里面顺子至少要 5 张连续的牌，我们这道题只计算长度最小为 3 的子序列，怎么办?

很简单，把我们的 else if 分支修改一下，连续判断 v 之后的连续 5 个元素就行了。

那么，我们再难为难为自己，如果我想要的不只是一个布尔值，我想要你给我把子序列都打印出来，怎么办?

其实这也很好实现，只要修改 need，不仅记录对某个元素的需求个数，而且记录具体是哪些子序列产生的需求：

```
// need[6] = 2 说明有两个子序列需要 6
unordered_map<int, int> need;

// need[6] = {
//     {3,4,5},
//     {2,3,4,5},
// }
```

```
// 记录哪两个子序列需要 6
unordered_map<int, vector<vector<int>>> need;
```

这样，我们稍微修改一下之前的代码就行了：

```
bool isPossible(vector<int>& nums) {
    unordered_map<int, int> freq;
    unordered_map<int, vector<vector<int>>> need;

    for (int v : nums) freq[v]++;

    for (int v : nums) {
        if (freq[v] == 0) {
            continue;
        }

        if (need.count(v) && need[v].size() > 0) {
            // v 可以接到之前的某个序列后面
            freq[v]--;
            // 随便取一个需要 v 的子序列
            vector<int> seq = need[v].back();
            need[v].pop_back();
            // 把 v 接到这个子序列后面
            seq.push_back(v);
            // 这个子序列的需求变成了 v + 1
            need[v + 1].push_back(seq);

        } else if (freq[v] > 0 && freq[v + 1] > 0 && freq[v + 2] > 0) {
            // 可以将 v 作为开头
            freq[v]--;
            freq[v + 1]--;
            freq[v + 2]--;
            // 新建一个长度为 3 的子序列 [v,v + 1,v + 2]
            vector<int> seq{v, v + 1, v + 2};
            // 对 v + 3 的需求加一
            need[v + 3].push_back(seq);

        } else {
            return false;
        }
    }

    // 打印切分出的所有子序列
    for (auto it : need) {
        for (vector<int>& seq : it.second) {
            for (int v : seq) {
                cout << v << " ";
            }
            cout << endl;
        }
    }
```

```
        return true;
    }
```

这样，我们记录具体子序列的需求也实现了。

如果本文对你有帮助，点个赞，微信会给你推荐更多相似文章。

如果你有高质量的问题，可以 [点击这里](#) 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号

如何判定完美矩形

Stars 103k | 知乎 @labuladong | 公众号 @labuladong | B站 @labuladong

微信搜一搜 labuladong公众号

学算法认准 labuladong，致力于把算法讲清楚！网页版 [点这里](#)。

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

[391. 完美矩形（困难）](#)

今天讲一道非常有意思，而且比较有难度的题目。

我们知道一个矩形有四个顶点，但是只要两个顶点的坐标就可以确定一个矩形了（比如左下角和右上角两个顶点坐标）。

今天来看看力扣第 391 题「[完美矩形](#)」，题目会给我们输入一个数组 `rectangles`，里面装着若干四元组 `(x1,y1,x2,y2)`，每个四元组就是记录一个矩形的左下角和右上角坐标。

也就是说，输入的 `rectangles` 数组实际上就是很多小矩形，题目要求我们输出一个布尔值，判断这些小矩形能否构成一个「完美矩形」。函数签名如下：

```
def isRectangleCover(rectangles: List[List[int]]) -> bool
```

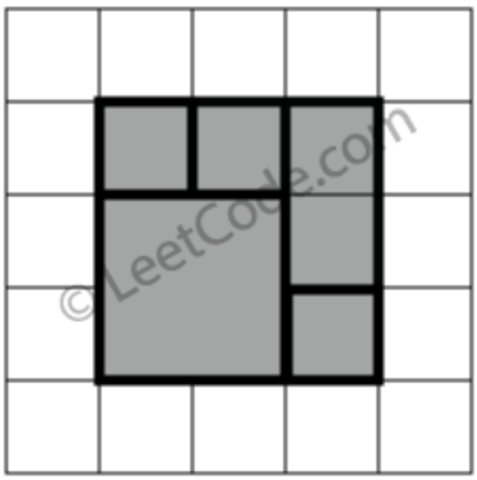
所谓「完美矩形」，就是说 `rectangles` 中的小矩形拼成图形必须是一个大矩形，且大矩形中不能有重叠和空缺。

比如说题目给我们举了几个例子：

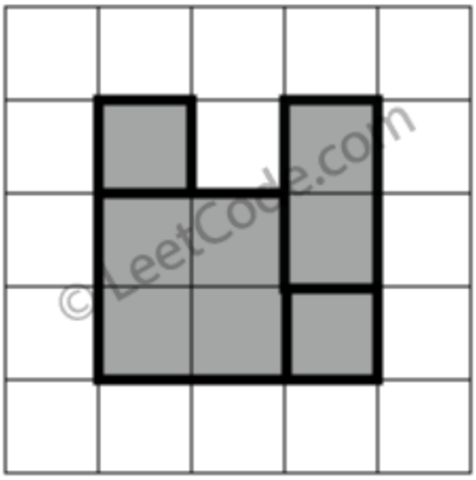
Example 1:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [3,2,4,4],
  [1,3,2,4],
  [2,3,3,4]
]

返回 true，因为最终形成的图形中没有空缺和重叠。
```



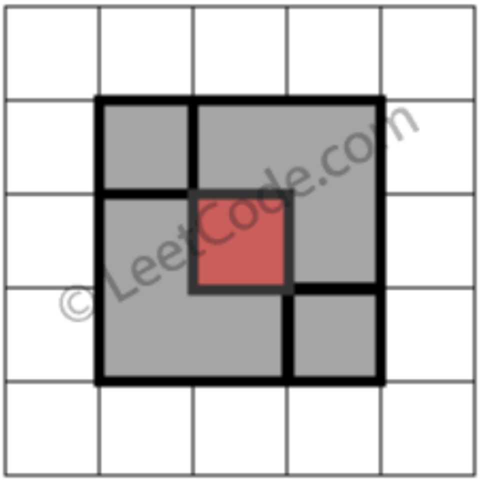
Example 2:




```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [3,2,4,4]
]

返回 false，因为最终形成的图形中有空缺。
```

Example 3:




```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [2,2,4,4]
]

返回 false，因为最终形成的图形存在重叠。
```

这个题目难度是 Hard，如果没有做过类似的题目，还真做不出来。

常规的思路，起码要把最终形成的图形表示出来吧，而且你要有方法去判断两个矩形是否有重叠，是否有空隙，虽然可以做到，不过感觉异常复杂。

其实，想判断最终形成的图形是否是完美矩形，需要从「面积」和「顶点」两个角度来处理。

先说说什么叫从「面积」的角度。

`rectangles` 数组中每个元素都是一个四元组 `(x1, y1, x2, y2)`，表示一个小矩形的左下角顶点坐标和右上角顶点坐标。

那么假设这些小矩形最终形成了一个「完美矩形」，你会不会求这个完美矩形的左下角顶点坐标 `(X1, Y1)` 和右上角顶点的坐标 `(X2, Y2)`？

这个很简单吧，左下角顶点 `(X1, Y1)` 就是 `rectangles` 中所有小矩形中最靠左下角的那个小矩形的左下角顶点；右上角顶点 `(X2, Y2)` 就是所有小矩形中最靠右上角的那个小矩形的右上角顶点。

注意我们用小写字母表示小矩形的坐标，大写字母表示最终形成的完美矩形的坐标，可以这样写代码：

```
# 左下角顶点，初始化为正无穷，以便记录最小值
X1, Y1 = float('inf'), float('inf')
```

```python
    # 右上角顶点，初始化为负无穷，以便记录最大值
    X2, Y2 = -float('inf'), -float('inf')

    for x1, y1, x2, y2 in rectangles:
        # 取小矩形左下角顶点的最小值
        X1, Y1 = min(X1, x1), min(Y1, y1)
        # 取小矩形右上角顶点的最大值
        X2, Y2 = max(X2, x2), max(Y2, y2)
```

这样就能求出完美矩形的左下角顶点坐标 `(X1, Y1)` 和右上角顶点的坐标 `(X2, Y2)` 了。

计算出的 `X1,Y1,X2,Y2` 坐标是完美矩形的「理论坐标」，如果所有小矩形的面积之和不等于这个完美矩形的理论面积，那么说明最终形成的图形肯定存在空缺或者重叠，肯定不是完美矩形。

代码可以进一步：

```python
def isRectangleCover(rectangles: List[List[int]]) -> bool:
    X1, Y1 = float('inf'), float('inf')
    X2, Y2 = -float('inf'), -float('inf')
    # 记录所有小矩形的面积之和
    actual_area = 0
    for x1, y1, x2, y2 in rectangles:
        # 计算完美矩形的理论坐标
        X1, Y1 = min(X1, x1), min(Y1, y1)
        X2, Y2 = max(X2, x2), max(Y2, y2)
        # 累加所有小矩形的面积
        actual_area += (x2 - x1) * (y2 - y1)

    # 计算完美矩形的理论面积
    expected_area = (X2 - X1) * (Y2 - Y1)
    # 面积应该相同
    if actual_area != expected_area:
        return False

    return True
```

这样，「面积」这个维度就完成了，思路其实不难，无非就是假设最终形成的图形是个完美矩形，然后比较面积是否相等，如果不相等的话说明最终形成的图形一定存在空缺或者重叠部分，不是完美矩形。

但是反过来说，如果面积相同，是否可以证明最终形成的图形是完美矩形，一定不存在空缺或者重叠？

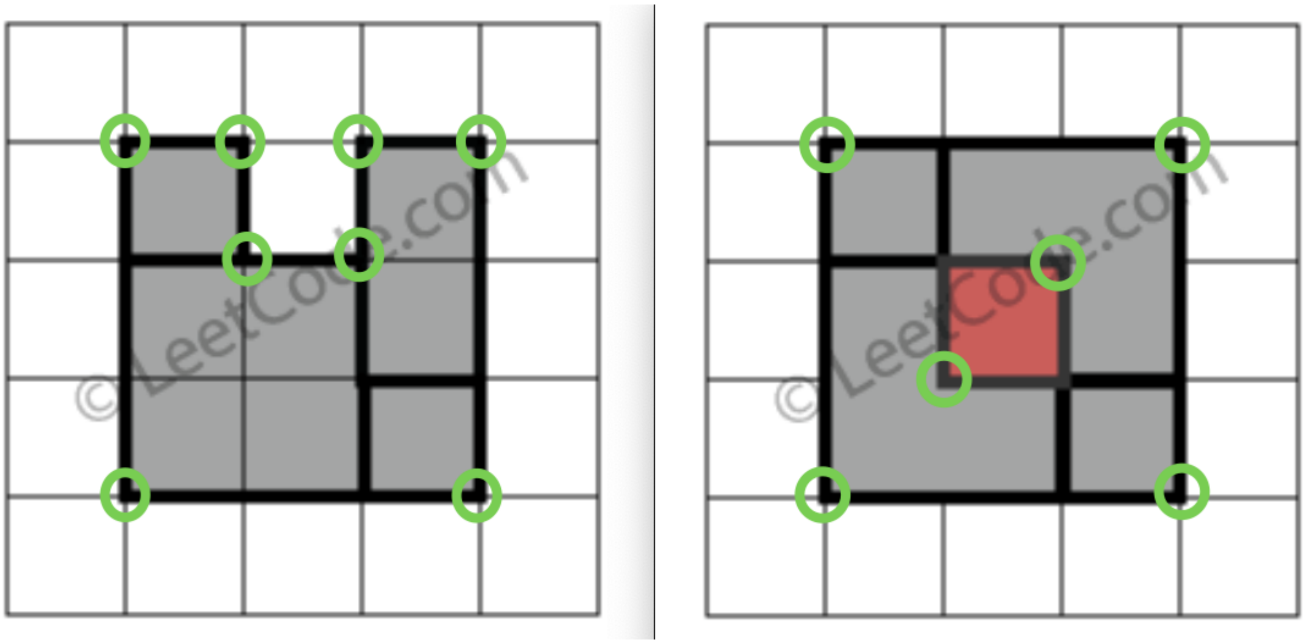
肯定是不行的，举个很简单的例子，你假想一个完美矩形，然后我在它中间挖掉一个小矩形，把这个小矩形向下平移一个单位。这样小矩形的面积之和没变，但是原来的完美矩形中就空缺了一部分，也重叠了一部分，已经不是完美矩形了。

综上，即便面积相同，并不能完全保证不存在空缺或者重叠，所以我们需要从「顶点」的维度来辅助判断。

记得小学的时候有一道智力题，给你一个矩形，切一刀，剩下的图形有几个顶点？答案是，如果沿着对角线切，就剩 3 个顶点；如果横着或者竖着切，剩 4 个顶点；如果只切掉一个小角，那么会出现 5 个顶点。

回到这道题，我们接下来的分析也有那么一点智力题的味道。

显然，完美矩形一定只有四个顶点。矩形嘛，按理说应该有四个顶点，如果存在空缺或者重叠的话，肯定不是四个顶点，比如说题目的这两个例子就有不止 4 个顶点：

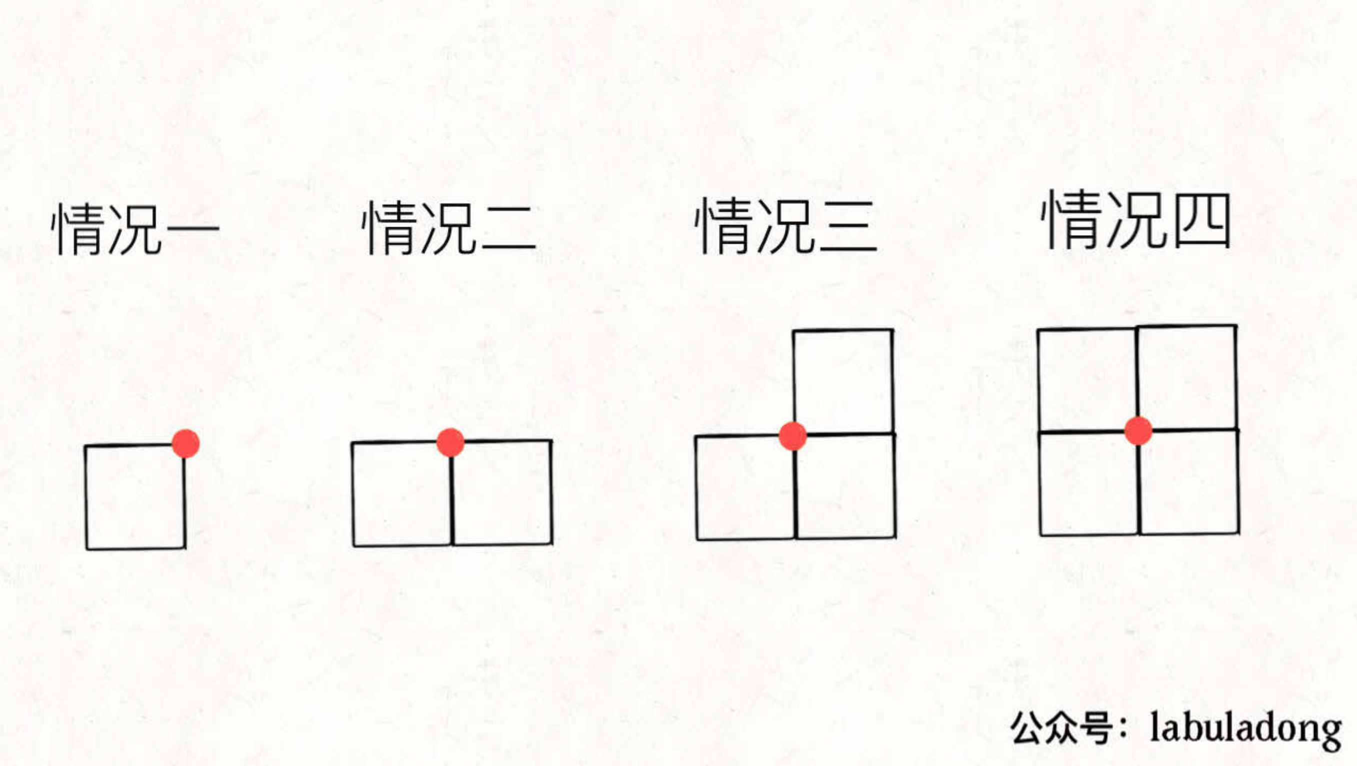


> PS：我也不知道应该用「顶点」还是「角」来形容，好像都不太准确，本文统一用「顶点」来形容，大家理解就好~

只要我们想办法计算 `rectangles` 中的小矩形最终形成的图形有几个顶点，就能判断最终的图形是不是一个完美矩形了。

那么顶点是如何形成的呢？我们倒是一眼就可以看出来顶点在哪里，问题是如何让计算机，让算法知道某一个点是不是顶点呢？这也是本题的难点所在。

看下图的四种情况：



图中画红点的地方，什么时候是顶点，什么时候不是顶点？显然，情况一和情况三的时候是顶点，而情况二和情况四的时候不是顶点。

也就是说，当某一个点同时是 2 个或者 4 个小矩形的顶点时，该点最终不是顶点；当某一个点同时是 1 个或者 3 个小矩形的顶点时，该点最终是一个顶点。

注意，2 和 4 都是偶数，1 和 3 都是奇数，我们想计算最终形成的图形中有几个顶点，也就是要筛选出那些出现了奇数次的顶点，可以这样写代码：

```
def isRectangleCover(rectangles: List[List[int]]) -> bool:
    X1, Y1 = float('inf'), float('inf')
    X2, Y2 = -float('inf'), -float('inf')

    actual_area = 0
    # 哈希集合，记录最终图形的顶点
    points = set()
    for x1, y1, x2, y2 in rectangles:
        X1, Y1 = min(X1, x1), min(Y1, y1)
        X2, Y2 = max(X2, x2), max(Y2, y2)

        actual_area += (x2 - x1) * (y2 - y1)
        # 先算出小矩形每个点的坐标
        p1, p2 = (x1, y1), (x1, y2)
        p3, p4 = (x2, y1), (x2, y2)
        # 对于每个点，如果存在集合中，删除它；
        # 如果不存在集合中，添加它；
        # 在集合中剩下的点都是出现奇数次的点
        for p in [p1, p2, p3, p4]:
            if p in points: points.remove(p)
            else: points.add(p)
```

```
    expected_area = (X2 - X1) * (Y2 - Y1)
    if actual_area != expected_area:
        return False

    return True
```

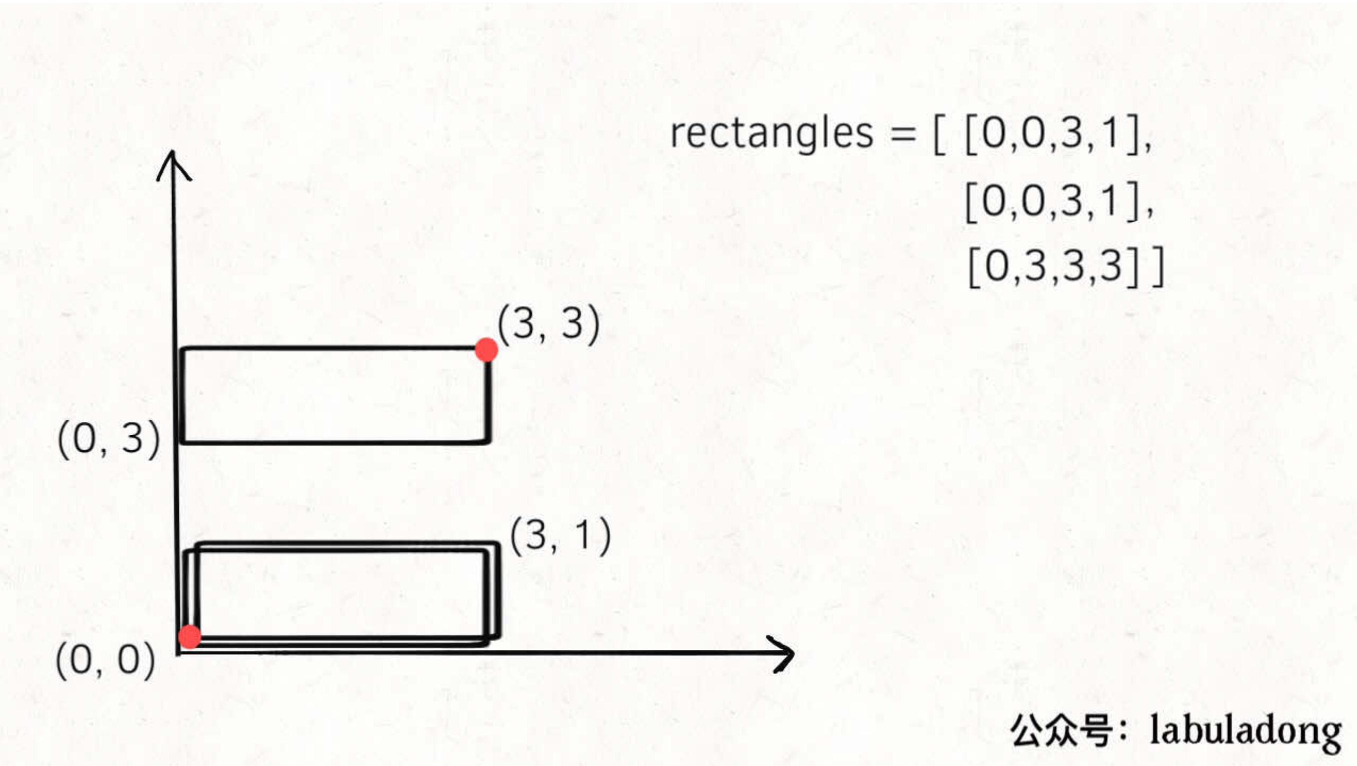
这段代码中，我们用一个 `points` 集合记录 `rectangles` 中小矩形组成的最终图形的顶点坐标，关键逻辑在于如何向 `points` 中添加坐标：

如果某一个顶点 `p` 存在于集合 `points` 中，则将它删除；如果不存在于集合 `points` 中，则将它插入。

这个简单的逻辑，让 `points` 集合最终只会留下那些出现了 1 次或者 3 次的顶点，那些出现了 2 次或者 4 次的顶点都被消掉了。

那么首先想到，`points` 集合中最后应该只有 4 个顶点对吧，如果 `len(points) != 4` 说明最终构成的图形肯定不是完美矩形。

但是如果 `len(points) == 4` 是否能说明最终构成的图形肯定是完美矩形呢？也不行，因为题目并没有说 `rectangles` 中的小矩形不存在重复，比如下面这种情况：



下面两个矩形重复了，按照我们的算法逻辑，它们的顶点都被消掉了，最终是剩下了四个顶点；再看面积，完美矩形的理论坐标是图中红色的点，计算出的理论面积和实际面积也相同。但是显然这种情况不是题目要求完美矩形。

所以不仅要保证 `len(points) == 4`，而且要保证 `points` 中最终剩下的点坐标就是完美矩形的四个理论坐标，直接看代码吧：

```
def isRectangleCover(rectangles: List[List[int]]) -> bool:
    X1, Y1 = float('inf'), float('inf')
    X2, Y2 = -float('inf'), -float('inf')
```

```python
    points = set()
    actual_area = 0
    for x1, y1, x2, y2 in rectangles:
        # 计算完美矩形的理论顶点坐标
        X1, Y1 = min(X1, x1), min(Y1, y1)
        X2, Y2 = max(X2, x2), max(Y2, y2)
        # 累加小矩形的面积
        actual_area += (x2 - x1) * (y2 - y1)
        # 记录最终形成的图形中的顶点
        p1, p2 = (x1, y1), (x1, y2)
        p3, p4 = (x2, y1), (x2, y2)
        for p in [p1, p2, p3, p4]:
            if p in points: points.remove(p)
            else:           points.add(p)
    # 判断面积是否相同
    expected_area = (X2 - X1) * (Y2 - Y1)
    if actual_area != expected_area:
        return False
    # 判断最终留下的顶点个数是否为 4
    if len(points) != 4:       return False
    # 判断留下的 4 个顶点是否是完美矩形的顶点
    if (X1, Y1) not in points: return False
    if (X1, Y2) not in points: return False
    if (X2, Y1) not in points: return False
    if (X2, Y2) not in points: return False
    # 面积和顶点都对应，说明矩形符合题意
    return True
```

这就是最终的解法代码，从「面积」和「顶点」两个维度来判断：

1、判断面积，通过完美矩形的理论坐标计算出一个理论面积，然后和 `rectangles` 中小矩形的实际面积和做对比。

2、判断顶点，`points` 集合中应该只剩下 4 个顶点且剩下的顶点必须都是完美矩形的理论顶点。

如果你有高质量的问题，可以 点击这里 进入本文的讨论区。

———————————————

关注公众号查看更多算法教程及训练营，后台回复关键词「进群」可进入刷题群，另《labuladong 的算法小抄》已经出版，公众号菜单查看优惠：

微信搜一搜
labuladong公众号